VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace modelů vytvořených v e-PFL pomocí UML diagramů

Visualization of models created in e-PFL using UML diagrams

2012 Bc. Tomáš Huplík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Huplík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Vizualizace modelů vytvořených v e-PFL pomocí UML diagramů
Visualization of Models Created in e-PFL Using UML Diagrams

Zásady pro vypracování:

Vestavný procesně funkcionální jazyk (Embedded Process Functional Language – e-PFL) je hybridní funkcionální jazyk určený pro modelování vestavných systémů v prvních fázích vývoje. Hlavním výstupem diplomové práce bude realizace nástroje, které umožní generovat UML diagramy popisující model vestavného systému z kódů v e-PFL. Cíle práce lze shrnout v těchto bodech:
1. Prozkoumejte možnosti technologie UML pro popis vestavných systémů.
2. Prozkoumejte možnosti jak pracovat s UML diagramy. Zaměřte se na vhodný formát (adekvátní se jeví XMI) a způsob jejich uložení, anebo vhodný nástroj pro jejich zobrazování.
3. Vyberte vhodné části modelu vestavného systému realizovaného v e-PFL, které bude možné a přínosné graficky znázornit pomocí UML diagramů.
4. Navrhněte a prakticky realizujte nástroj, který z modelu vestavného systému v e-PFL vytvoří vybranou množinu UML diagramů.
5. Na případových studiích demonstrujte možnosti vytvořeného nástroje.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 19.11.2010
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012 .....

Touto cestou bych rád poděkoval vedoucímu diplomové práce, panu Ing. Marku Běhálkovi, Ph.D., především za konzultace a odborné rady, které jsem zúročil při psaní tohoto textu.

Abstrakt

Hlavním výstupem diplomové práce je realizace nástroje, umožňujícího generovat UML diagramy ze systému, který je popsán ve vestavném procesně funkcionálním jazyce e-PFL. Úkolem těchto diagramů je zpřehlednit a usnadnit pochopení navrhovaného systému. Vlastní nástroj je integrován do stávajícího překladače e-PFL a generuje validní XMI struktury souborů, které mohou být dále zpracovány v CASE nástrojích do podoby konkrétních diagramů. V teoretické části jsou obecně popsány charakteristiky vestavěných systémů, možnosti jejich modelování a využití technologie UML pro tuto oblast. Součástí je také popis samotného jazyka e-PFL a standardu XMI, jakožto klíčového standardu pro výměnu objektů UML modelů a metadat informací. Praktická část se zaměřuje na návrh a realizaci vlastního nástroje a jeho integraci s nástrojem e-PFL. Jsou zde přiloženy i příklady použití a případová studie demonstrující funkčnost na konkrétní úloze v e-PFL.

Klíčová slova: diplomová práce, vestavěné systémy, UML, XML Metadata Interchange, XMI, vestavěný procesně-funkcionální jazyk, e-PFL, CASE nástroje

Abstract

The aim of this thesis is the implementation of a tool which enables the generation of UML diagrams from the system which is described in e-PFL, the embedded process-functional language. The diagrams are to provide transparency and better understanding of the suggested system. The tool itself is integrated into an existing compiler of the e-PFL. It generates valid XMI structures of files which may be further processed and formed into specific diagrams using the CASE tools. The theoretical part of the thesis describes general characteristics of the embedded systems, their possible simulation and the usage of the UML technology for this area. Furthermore, this part describes the e-PFL language itself and the XMI standard as a key standard for the interchange of the UML objects of models and metadata information by the XML. The practical part of the thesis focuses on the plan and realization of the tool and the integration of the tool with the e-PFL tool. Additionally, this part contains examples of usage and a case study in which the functionality is demonstrated on a specific problem in e-PFL.

Keywords: Master's thesis, embedded systems, UML, XML Metadata Interchange, XMI, embedded process-functional language, e-PFL, CASE tools

Seznam použitých zkratek a symbolů

CASE	–	Computer-aided Software Engineering
CMOF	–	Complete MOF
CWM	–	Common Warehouse Metamodel
DP	–	Diplomová Práce
DTD	–	Document Type Definition
e-$\mathcal{PFL}$	–	Embedded Process Functional Language
EMF	–	Eclipse Modeling Framework
EMOF	–	Essential MOF
FAQ	–	Frequently Asked Questions
HDL	–	Hardware Description Language
MARTE	–	Modeling and Analysis of Real Time and Embedded systems
MOF	–	Model Driven Architecture
OCL	–	Object Constraint Language
OMG	–	Object Management Group
OOP	–	Object-Oriented Programming
$\mathcal{PPFL}$	–	Parallel Process Functional Language
RSA	–	Rational Software Architect (IBM)
RT-UML	–	Real-Time UML
RTOS	–	Real-Time Operating System
SPEM	–	Software Process Engineering Metamodel
SysML	–	Systems Modeling Language
UML	–	Unified Modeling Language
XMI	–	XML Metadata Interchange
XML	–	Extensible Markup Language

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Předložená diplomová práce volně navazuje na disertační práci „Vestavěný procesně funkcionální jazyk e-*PFL*“ [4], kde autor Ing. Marek Běhálek, Ph.D. (který je rovněž vedoucím této diplomové práce) v kapitole 7.2 shrnuje možnosti dalšího výzkumu mimo jiné o vizualizaci vytvářených modelů v jazyce e-*PFL*.

Diplomová práce si klade za cíl v obecné rovině seznámit s možnostmi využití UML pro embedded systémy a prozkoumat některé z dostupných prostředků pro přenos a vizualizaci UML modelů. Nedílnou součástí je také návrh a praktická realizace nástroje, který bude schopen generovat UML modely v adekvátním souborovém formátu. Tyto modely mohou být později načteny a zpracovány nástroji pro vizualizaci diagramů (CASE tools). Realizovaný nástroj bude integrován do e-*PFL* tak, aby bylo možné automatizovaně generovat diagramy z popsaného vestavěného systému a zvýšila se tak jeho srozumitelnost. Výsledný nástroj bude demostrován na příkladech a případových studiích ve spojení s e-*PFL*.

V druhé kapitole jsou stručně popsány související technologie a přístupy, které je potřeba znát k pochopení a naplnění cílů zadání této diplomové práce. První část se věnuje obecně pojmu „vestavěný systém“, jeho charakteristikami a dále pak možnostmi modelování takovýchto systémů. V druhé části jsou popsány klíčové vlastnosti jazyka UML, jeho struktura, meta modelování, možnosti rozšíření (včetně uvedení některých profilů) a popis jednotlivých hledisek využití v oblasti vestavěných systémů. Kapitola dále popisuje standard *XML Metadata Interchange*, určený pro výměnu objektů modelu a metadat informací pomocí XML. Na tomto standardu byl realizován vlastní nástroj, který je hlavním výstupem diplomové práce. V poslední části je stručně popsán význam a vlastnosti CASE nástrojů.

Třetí kapitola se zaměřuje na samotný „Vestavěný procesně funkcionální jazyk“ e-*PFL*. Popisuje jeho klíčové vlastnosti, strukturu, syntaxi a také související nástroje pro překlad a simulaci e-*PFL* zdrojového kódu.

Čtvrtá kapitola obsahuje teoretické východiska a nástin řešené problematiky tak, jak byly chronologicky řešeny. V této části je rovněž zdůvodněn výběr prostředků, nástrojů a technologií a také popis vhodných částí modelu vestavného systému realizovaného v e-*PFL*, které je přínosné graficky znázornit pomocí UML diagramů.

Pátá kapitola popisuje návrh a realizaci vlastního nástroje, včetně integrace s e-*PFL*. Jsou zde i příklady konstrukce UML elementů, včetně ukázky struktury výstupních XMI souborů a diagramů vytvořených z těchto souborů. Na konci kapitoly je uvedena případová studie, kde je demonstrována spolupráce s e-*PFL* a to včetně popisu a ukázek generovaných diagramů.

Závěrečná šestá kapitola shrnuje dosažené výsledky a přínos této diplomové práce, dále definuje případné možnosti rozšíření práce do budoucna.

2 Související technologie a přístupy

2.1 Vestavěné systémy obecně

2.1.1 Základní charakteristika a vlastnosti

Vestavěné systémy (embedded systems) [5, 24, 2] jsou počítačové systémy navržené pro vykonávání jedné, nebo několika vyhrazených funkcí. Jeden ze základních požadavků na tyto systémy je, že musí být tzv. „autonomní“ čili schopny plnit své funkce bez zásahu člověka v průběhu poměrně dlouhého časového intervalu. Proto je při vývoji kladen hlavní důraz na energetickou spotřebu, spolehlivost a robustnost. Autonomní činnost je nezbytná především tam, kde reakce člověka mohou být příliš pomalé, nedostatečně předvídatelné nebo nežádoucí. Systémy, které pracují v reálném čase, musí být velmi rychlé při vykonávání daných funkcí. Embedded systémy jsou pro uživatele neviditelné, proto by je uživatel neměl považovat za plnohodnotný počítač. Software určený pro vestavěné systémy je často označován jako firmware a je uložen v čipech ROM nebo Flash. Mezi oblasti využití patří mimo jiné např. mobilní telefony, digitální kamery a foto přístroje, chytrá domácí elektronika, systémy pro medicínu, měřicí přístroje, armádní systémy, zařízení v dopravním a leteckém průmyslu (řídící jednotky, navigační zařízení), telekomunikační a síťové prvky (ústředny, router, switch, firewall), bankomaty atd.

2.1.2 Modelování a vývoj vestavěných systémů

Návrh a vývoj softwarového systému je komplexní a poměrně složitá disciplína. Speciálně u real-time a embedded systémů, které mají své specifické požadavky při návrhu. Při chybném návrhu softwaru vestavěného zařízení, nejen že může dojít k fatálním následkům (i životně a majetkově ohrožující), ale často není také jednouché případnou chybu v návrhu a implementaci napravit. Zařízení jsou zpravidla vyráběny velkosériově a následná aktualizace firmware všech kusů je stěží dosažitelná, drahá, nebo i nemožná. Je zcela evidentní, že už v rané fázi vývoje je zapotřebí kvalitně zachytit, namodelovat a zdokumentovat, celou funkčnost takových systémů.

Z pohledu vývojáře můžeme shrnout některá fakta [2]:

- Hardware i software jsou vyvíjeny paralelně.
- Pro HW konstrukci je na trhu velké množství různých typů mikroprocesorů.
- K dispozici jsou různé operační systémy, přičemž se ve vestavných aplikacích převážně používají operační systémy reálného času.
- Disponují menším množstvím systémových zdrojů, než je tomu u stolních systémů (např. menší kapacitou paměti).
- K vývoji jsou nutné vesměs dosti nákladné speciální vývojové nástroje jako emulátory, logické analyzátory, speciální překladače vyšších programovacích jazyků se zaručenou spolehlivostí generovaného kódu apod.

- Ladění programového vybavení je podstatně složitější než u programů pro PC.

Proces modelování vestavěných systémů zahrnuje tyto fáze:

1. Specifikace systému
2. Rozložení na HW a SW část
3. Hardware syntéza
4. Software syntéza, generování kódu
5. Simulace a verifikace
6. Implementace
7. Testování

Mezi hardwarové specifikační nebo modelovací prostředky patří některé jazyky z rodiny HDL (hardware description language) [25] např. *VHDL*, *Verilog*, *Lava (Haskell)*, *SystemC* atd. Další používaným prostředkem pro vestavěné systémy je komerční nástroj *Simulink*, který je určen pro modelování, simulaci a analýzu dynamických procesů a je úzce integrován s Matlab. Mezi skupinu prostředků založených na anotaci UML modelů patří např. *SysML*, *MARTE*, *RT-UML*, *Embedded UML*, *UML state machine* atd.

Co se týče specializovaných nástrojů pro grafický návrh různých diagramů popisující úlohu, generování dokumentace, podporující týmovou spolupráci apod., existuje celá řada společností zabývající se tímto odvětvím jako např. *Rhapsody od firmy I-Logix*, nebo *Embedded Rational Rose* od firmy IBM. Rhapsody má také unikátní schopnost rozšířit UML o možnost používat jak funkcionálně orientované, tak objektově orientované návrhové techniky a spoluexistovat v jednom prostředí.

2.2 Unified Modeling Language

2.2.1 Popis a struktura

UML (Unified Modeling Language) [9, 7, 1, 10, 15] je v softwarovém inženýrství velmi komplexní grafický jazyk pro analýzu, vizualizaci, specifikaci, navrhování a dokumentaci softwarových systémů a to především vytvořených pomocí objektově orientovaného přístupu. Tento unifikovaný modelovací jazyk lze rovněž využít při modelování business procesů, databázových schémat, programových komponent, konceptuálních schémat atd. Cílem je zaznamenat kompletní návrh či kompletní realizaci zainteresovanými osobami zejména v grafické formě tak, aby byly poměrně jednoduché na čtení, či pochopení a aby mohly být případné změny provedeny s poměrně malým úsilím a znalostmi. UML je standardizován a jeho formální specifikace je pod vedením standardizační skupiny Object Management Group[1] (OMG). V současné době existuje specifikace verze 2.4.1.
UML nepředepisuje přesně jak postupovat při analýze pomocí diagramů, ale literatura [10] uvádí tři běžné postupy:

[1] http://www.uml.org/

1. **Sketches (náčrty)** - jsou neformální diagramy, které slouží k zachycení myšlenek, hledání alternativ, nebo k společnému návrhu designu. Dělají se ručně, nejsou detailní a jsou zaměřeny na určíté aspekty systému.

2. **Blueprint** - jsou detailních návrhy tak, aby bylo možné systém bezchybně implementovat. Využívají se různé podpůrné SW nástroje (CASE nástroje) pro vytvoření diagramů, generování zdrojových kódu z diagramu, nebo generování diagramů ze zdrojových kódu (reverse engineering).

3. **Použití UML jako programovacího jazyka** - pokud je použita tato forma, celý systém je specifikován v UML, diagramy jsou „zdrojovým kódem" a jsou kompilovány přímo do spustitelných binárních souborů. UML nástroj nám tak umožní vytvořený model přímo spustit. Tento přístup úzce souvisí s Exeutable UML a MDA.

Formální specifikace UML je rozdělena do těchto skupin:

- **UML Superstructure (syntaxe)** [20] - obsahuje diagramy struktury a jejich chování.
- **UML Infrastructure (sémantika)** [21] - definuje jádro (core) obsahující meta-třídy, nad kterými je možné vystavět meta-modely. Ve vztahu k UML to je meta-model UML (ze specifikace Superstructure), MOF (Meta Object Facility), nebo meta-model UML profilu. Kromě toho nad těmi samými meta-třídami jsou postaveny i elementy samotné knihovny Infrastructure; říkáme, že je reflexivní.
- **UML Object Constraint Language (OCL)** [18] - jazyk používaný ke specifikaci omezení nad elementy UML.
- **UML Diagram Interchange** - definuje formát přenosu UML struktur jako jsou CORBA Interface Definition Language, XML DTD, XMI (XML Metadata Interchange).

Jednotlivé UML diagramy se dělí do těchto skupin (pro kompletní výpis a podrobný popis odkazuji na literaturu [9, 15, 21]):

- **Strukturní diagramy** - diagram tříd, diagram komponent, kompozitní strukturální diagram, diagram nasazení, diagram balíčků, diagram objektů (diagram instancí).
- **Diagramy chování** - diagram aktivit, diagram užití, stavový diagram.
- **Diagramy interakce** - sekvenční diagram, diagram komunikace, interaction overview diagram, diagram časování.

UML popisuje systém z pohledu různých zainteresovaných stran jako je např. koncový uživatel, vývojář, projektový manažer atd. Poskytnutí srozumitelného pohledu na navrhované řešení pro všechny role přináší tzv. ***4+1 pohled na architekturu***:

1. **Logický pohled** - týká se funkcí, které systém poskytuje koncovým uživatelům. Zahrnuje statické struktury, ale i dynamické chování systému (diagramy tříd, sekvenční diagramy, diagramy spolupráce, stavové diagramy).

2. **Implementační pohled** - pohled zachycuje organizaci kódu a skutečného provedení kódu (diagram komponent, diagram balíčků).

3. **Procesní pohled** - zachycuje dynamické vlastnosti procesů, ukazuje hlavní prvky v systému týkající se vzájemné komunikaci procesů a chování systému za běhu. Tento pohled zachycuje souběžnosti, distribuci, integrátory, škálovatelnost, výkon, dostupnost (diagram aktivit).

4. **Pohled nasazení** - zobrazení zavedení systému do fyzického architektura s počítači a zařízeními tzv. uzly (diagram nasazení).

5. **Případy užití** - skládá se ze souboru případů užití nebo scénářů. Scénáře popisují sekvence interakcí mezi objekty a mezi procesy. Používá se k identifikaci prvků architektury a pro ilustraci a potvrzení architektury. Zároveň slouží jako výchozí bod pro testy prototypu architektury.

2.2.2 MOF, Meta model, čtyřvrstvá architektura

MOF (Meta Object Facility)[16] je OMG standard poskytující platformě nezávislý rámec pro správu metadat a množinu metadat služeb, umožňující vývoj a interoperabilitu modelem a metadaty řízenými systémy. MOF byl navržen pro potřebu meta-modelování UML. MOF a UML Infrastructure jsou bohaté a komplexní metamodelovací jazyky a mohou být použity k definování modelovacích jazyků tak rozsáhlých a složitých, jako je UML. Mohou být také použity k definování jazyků pro konkrétní domény, nebo jako rozšíření a profily pro UML.

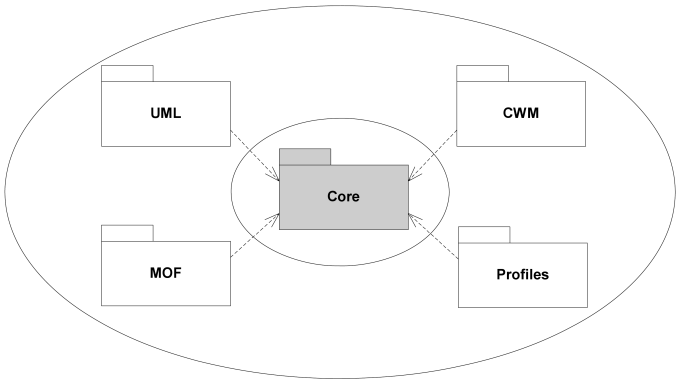
Tyto metamodelovací jazyky mohou popsat pouze strukturu, či abstraktní syntaxi (nikoliv konkrétní) modelovacího jazyka. Dovolují definování prvků modelu a vztahů mezi nimi, ale ne jejich vizuální reprezentaci. Z důvodu podobnosti mezi MOF M3 modelem a UML strukturalním modelem, MOF metamodely jsou obvykle modelovány jako UML class diagramy.

Technologie standartizované rovněž OMG jako např. UML, CWM, SPEM, XMI a různé UML profily využívají MOF a MOF řízené technologie. Jeden z důležitých standartů pro podporu MOF je XMI, který definuje podporu výměny modelů na M3, M2 a M1 vrstvě prostřednictvím XML formátu.

MOF 2.0 se skládá ze dvou částí [20, 16]:

1. EMOF (Essential MOF) - je minimální podmnožina MOF a slučuje elementy z balíku Basic.

2. CMOF (Complete MOF) - je kompletní množina MOF, obsahuje EMOF a navíc rozšiřuje modely o další schopnosti.



Obrázek 1: Balík Core (Infrastructure Library) [20]

MOF je složený z čtyřvrstvé architektury a poskytuje meta-model na nejvyšší vrstvě zvané M3. Nad touto vrstvou se popisují prostředky pro vytváření a manipulaci s modely a metamodely. Je nutno poznamenat, že MOF 1.0 a MOF 2.0 dovoluje použít jakýkoli počet vrstev, ale nejméně musí být dvě. To proto, abych mohly znázorňovat nějaké třídy a navigovat se na její instanci a naopak. Zde je přehled jednotlivých vrstev:

- **M3-vrstva**: na nejvyšší úrovni je mata-meta model zvaný M3. M3-model je jazykem používaným MOF pro konstrukci metamodelů zvaných M2-modely.
- **M2-vrstva**: zde je M2-model, který tvoří popis elementů ve vrstvě M1 a tedy M1-modely. Hlavní příklad druhé vrstvy MOF modelu je UML metamodel, neboli model, který popisuje vlastní UML.
- **M1-vrstva**: zde je M1-model, popisuje modely vytvořené v UML.
- **M0-vrstva**: poslední vrstva je M0, nebo také datová vrstva. Využíva se pro popis objektů reálného světa.

2.2.3 Možnost rozšíření UML

Ačkoli je UML do jisté míry aplikačně a jazykově nezávislý, potřebujeme někdy tento jazyk rozšířit o další sémantiku pro specifickou doménu problémů [20, 9]. Předdefinovaná množina mechanismu rozšíření tzv. extension se nazývá profil. Profil nerozšiřuje UML vytvořením nového konceptu, neboli neporušuje existující principy modelování. Díky tohoto principu si UML extenze zachovává jednoduchost a nijak se neodklání od původního konceptu. Na druhou stranu nelze odstranit existující omezení.

Důvody, které nás vedou k rozšíření, mohou být tyto:

- přidání nové sémantiky (významu);
- přidání nové syntaxe pro neexistující elementy v UML;

- vytvoření, nebo změna vlastností;
- propojení terminologie domény s modelem systému;
- omezení meta-modelu tak, abychom povolili pouze určitý způsob nakládání s elementem modelu.

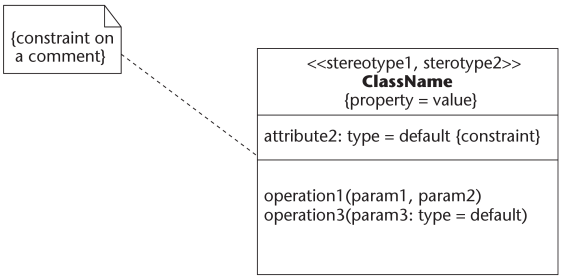
Rozšíření UML můžeme realizovat takto:

1. Vytvoření vlastních elementů meta-modelu pomocí MOF, bez použití stávajících elementu metamodelu UML - vytvoření úpně nového modelovacího jazyka, jsou s tím spojeny značné ryzika a úsilí, znamená také vytvořit vlastní modelovací nástroje.
2. Využití části meta-modelu UML a rozšíření pomocí MOF - nekompatibilní se stávajícími nástoji, musí se přidat nové možnosti.
3. Využít balík *Profile* a postavit rozšíření nad celým UML - nejobvyklejší způsob se zachováním kompatibility s UML. Využívá se ve většině případů.

Standardní prvky rozšíření UML („Lightweight“ rozšíření sémantiky):

- **Stereotypes** - stereotypy jsou tzv. „typy typů“ nebo „meta-typů“. To znamená, že mohou být použity ke klasifikaci téměř všech elementů v UML jako jsou třídy, uzly, komponenty, balíčky, vztahy, sdružení, generalizace a závislosti. Stereotypy umožňují klasifikovat elementy diagramů a tím vyjádřit další sémantiku s cílem definovat další element modelu. Můžeme to chápat tak, že přidávají význam elementu v UML, aby přesněji popisoval roli elementu v rámci našeho modelu. Řada stereotypů je již předdefinovaná v UML a jsou používány k úpravě existujícího modelu elementů namísto definování stereotypu nového. Tato strategie udržuje základní jazyk UML jednoduchý. Jako příklad lze uvést např. základní stereotyp rozhraní «*interface*».
- **Tagged values** - označení hodnot umožňuje k atributům prvků nastavit další informace k udržení informací o elementech. Může se jednat o libovolný typ informací, které chce uživatel připojit jako specifické informace pro metody, administrativní informace o průběhu modelování, informace pro nástroje pro generování kódu apod. Každý stereotyp muže mít svou množinu „tagged values“, které jsou odděleny od standardního atributu elementu, tudíž nedochází k jejich překrytí.
- **Constraints** - omezuje použití prvku nebo sémantiku prvku [18]. Je to soubor pravidel vyjádřený v OCL nebo v přirozeném jazyce. Omezení jsou prostředky, kterými můžeme zavést novou sémantika do UML.

Celá řada UML profilů byla navržena na akademických půdách, nebo v průmyslu, přičemž některé z nich byly i standardizovány. Jako příklad můžeme uvést tyto UML profily: *Profile for CORBA*; *UML Profile for QoS and Fault Tolerance*; *UML Profile for Schedulability, Performance, and Time*; *UML Profile for System on a Chip (SoCP)*. Budeme-li se



Obrázek 2: Příklad použití stereotypes, tagged values a constraints [9]

bavit o profilech, které byly adaptovány na specifické vlastnosti vestavěných (real-time) systémů, je tady především standardizovaný profil *MARTE*[2]. Jako další můžeme uvést např. *RT-UML*, *TUT-Profile*, *OMEGA-RT*, *SDL-UML profile*, *SPT profile*, *SystemC*, *Rhapsody* atd.

2.3 Použití UML pro vestavěné systémy

Snaha organizací o úspěšný týmový vývoj komplexních vestavěných aplikací přivedla k použití mnoha nástrojů a prostředků, mezi nimiž je také bezesporu velmi populární UML [7, 6]. Ačkoli je UML primárně určen pro modelování objektově orientovaného (nebo tomuto paradigmatu blízkého) systému, lze jej s úspěchem využít také pro modelování vestavěných resp. real-time systémů. UML 2 zavádí notaci portů, stejně jako protokol stavových automatů, které architekt využívá pro zobrazení vztahů tříd k jejím prostředí a chování. Tyto nové vlastnosti podporované objektově orientovanými metodami specificky zaměřenými na vestavěné resp. real-time systémy nejsou dobře podporovány v UML 1.x. Konsorcium OMG každou novou specifikací UML podporu pro modelování vestavěných systémů zlepšuje. Existuje i celá řada UML profilů, které rozšiřují UML o specifickou sémantiku zaměřenou na real-time systémy, jako už zmiňovaný MARTE.

Základní výhody použití UML pro vestavěné systémy můžeme shrnout v těchto bodech [13]:

- hardwarová/softwarová a částečná technologická nezávislost;
- umožňuje zachytit funkční i nefunkční požadavky a omezení;
- umožňuje vysoce modulární systémový design;
- umožňuje získání celkového pohledu na systém;
- pomoc při vizualizaci, určení, konstrukce a dokumentaci hardware/software systému;
- srozumitelnost - reference na vlastní sémantiku;

[2] http://www.omgmarte.org

- kontrola konzistence - realizace vs. design, design vs. specifikace;
- lze snadno přidávat nové charakteristické rysy

Zajisté jsou známy i určité nevýhody [13]:

- formalizace jsou těžko řešitelná pro neodborné uživatele;
- příliš podrobné formalizace mohou vytvořit zmatek mezi zúčastněnými stranami projektu;
- není možno formalizovat na nižší úrovni v případě potřeby;
- chybí definice omezení a rozpočtová metodika k doplnění přesunu designu z požadavků až k implementaci, které zajišťují nezbytnou kontrolu pro optimalizaci procesů v transformaci návrhu a syntézy;
- nedostatečná metodika mapování a zjemnění přechodu z jedné úrovně platformy na jinou úroveň

Stejně jako všechny ostatní sytémy, mají i vestavěné systémy statickou strukturu reprezentovanou pomocí tříd a objektů, které jsou navzájem spojené pomocí asociací, generalizace a závislostí. Vestavěný systém má rovněž nějáké chování, které můžeme popsat pomocí dynamických modelů, pohledů vývoje a nasazení, které lze modelovat pomocí komponent a diagramů nasazení.

Při samotném modelování se zaměřujeme na tyto oblasti [9]:

- **Aktivní třída a aktivní objekt** - je klíčová struktura real-time systémů. Aktivní třídy vytvářejí aktivní objekty, které mají kontrolu nad prováděním (spouští a řídí své vlastní vlákno). Aktivní objekt může být vykonáván paralelně s ostatními aktivními objekty a sám o sobě může inicializovat akce vlastním kódem, nebo posláním zpráv. Aktivní třída reprezentuje jednotku, která je vykonávána konkurenčně. Naproti pasivní třída má instance, které mohou být vykonávány jen když nějáký jiný objekt vykonává nějakou operaci na něm. Aktivní objekt potřebuje více systemových prostředků, než pasivní objekty.
- **Komunikace** - UML používá zprávy (messages), spouště (triggers) a signály (signals) pro modelování synchronní a asynchronní komunikace. Pokročilé dynamické diagramy potřebují zobrazit typ zprávy, poslanou mezi vláknem, nebo aktivovanou z asynchronní události, nebo signálu v prostředí.
- **Synchronizace** - synchronizace koordinuje vykonávání vláken. Systémem požadovaná synchronizace vyvolává řadu otázek. Architekt musí přijít s mechanismy pro řešení těchto otázek, jako i s definováním priorit pomocí „tagged values“, nebo s hlídacímy mechanismy pro kritické oblasti.

- **Odolnost vůči chybám** - strategie komplexní komunikace a synchronizace vestavěných systémů závisí na dobrém ošetření chyb (exception handling). Systém by se měl sám efektivně napravit ze všech potencionálních chybových situací.

Využití UML diagramů pro vestavěné systémy:

- **USE-CASE diagrams** - odkazují na chování skrz skupinu aktivit. Užití toho diagramu se zajistí, že dílo bude ve stanoveném rozsahu, ale už nemá schopnost popsat detailní toky systémem.
- **Activity diagrams** - poskytují vysokoúrovňový pohled na systém, který není vázaný na systémové klasifikátory. Používá se při zobrazení toků v sytému, i pro případy odehrávající se vně systému který navrhujeme. Diagram může poskytovat užitečnou mapu pro všechny objekty komunikující s tímto projektem. Navíc může diagram pomoct v definování doménových entit se vstupními a výstupními body, které mohou být spojeny s elementy reálného sytému.
- **Object diagrams and internal structure diagrams** - může poskytovat důležité informace o tom, které třídy mají aktivní chování. Tyto diagramy poskytují požadované vstupy pro interakční diagramy a stavové automaty.
- **Interaction diagrams** - mohou zobrazit detailní zprávy, které se vyskytují ve specifické akci, nebo skupině akcí. Takový detail se dostává velmi blízko k úrovni kódu a poskytuje dobrou cestu ke interakční komunikaci mezi objekty. Použití je pro přehled záležitostí blízkých implementovanému kódu. Může se zde propojit sekvence s nějakou akcí systému, což dává snadnou možnost přecházet z diagramů aktivit až k sekvenčním diagramům.
- **State machine** - ukazují komplexní životní cyklus objektů. Zjískáte tak přehled aktivních objektů, nebo interní životnost více komplikovaných objektů systému. Někdy se užívají stavové automaty jako základ pro generování kódu.
- **Protocol state machine** - zobrazuje jak nějaká entita nespoléhá jen na prostředí, ale také na současné okolnosti k jeji vyřízení. Použitím zobrazíme jak komponenty pracují s porty v distribuovaném prostředí.

Harware může být zachycen skrz „wrapped“ třídy, které jsou navrženy k prezentaci rozhraní hardware a požadavky hardware. Porty poskytují modelovací elementy, ukazující vazbu mezi klasifikátorem a harwarovým prostředím. Tyto „wrappery“ zřizují komunikační protokol použitý s zařízením a přerušením, která může zařízení generovat. Použití takových HW tříd umožňuje skrýt nízkoúrovňový protokol a přerušení na nízké úrovni a přeloží je na vysokoúrovňové události (events) ve zbytku systému. Distribuční mechanismus zahrnuje serializaci a deserializaci komplexních objektů, které mohou být poslány na komunikační lince. To také zahrnuje úkol, kterým se zavádí globální objekt identifikující system tak, aby objekty mohly být adresovány mezi sebou, aníž by věděly přesné umístění. Zde musí být také serverové procesy nebo vlákna k přístupnění objektů registru a vyřešit požadavek na tento objekt v provedení aktuální operace na objektu.

2.3.1 UML profil MARTE

Tato specifikace [19] UML profilu rozšiřuje UML o modelem řízený vývoj real-time a vestavěných systémů (RTES). Jedná se v současné době o nejkomplexnější profil pro tyto účely. Poskytuje podporu během etap specifikace, návrhu a verifikace/validace. MARTE[3] spočívá v definování základů pro popis vestavěných systémů a systémů v realném čase založených na modelech. Modelovací části nabízí podporu požadovanou od zadání, až po detailní návrh real-time a embedded vlastností systémů. Nebylo záměrem definovat nové technologie pro analýzu RTES, ale pouze zavést jejich podporu. Proto poskytuje zázemí pro komentované modely s informacemi, potřebnými k provedení konkrétní analýzy. MARTE se zvlášť zaměřuje na „performance and schedulability analýzu". Specifikace je vedena pod OMG konsorciem a v současné době je dostupná verze MARTE 1.1.

Mezi základní výhody použití tohoto profilu patří:

- Poskytnutí modelování běžným způsobem z hardwarových i softwarových aspektů RTES s cílem zlepšit komunikaci mezi vývojáři.
- Povolení interoperability mezi vývojovými nástroji pro specifikaci, design, ověřování, generování kódu, atd.
- Podpora výstavby modelů, které mohou být použity pro kvantitativní předpovědi týkajících se real-time a vestavěných funkcí systémů s ohledem na hardwarové i softwarové vlastnosti.

Na podporu modelování RT systémů nabízí MARTE tyto čtyři pilíře:

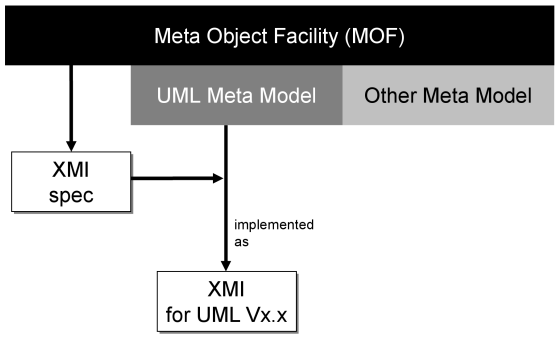
1. QoS-aware Modeling
    - HLAM (High-Level Application Modeling) - pro modelování na vysoké úrovni RT QoS, včetně kvantitativních a kvalitativních problémů.
    - NFP (Non-Functional Properties) - pro deklaraci, kvalifikaci a aplikování sémanticky správně zformulovaných nefunkcionálních požadavků.
    - Time (Enhanced Time Modeling) - pro definici času a manipulaci s jeho reprezentací.
    - VSL (Value Specification Language) - je textový jazyk pro specifikaci algebraických výrazů.
2. Architecture Modeling
    - GCM (Generic Component Model) - pro modelování architektury založené na interakci komponent pomocí zpráv nebo dat.
    - Alloc (Allocation Modeling) - pro určení rozdělení funkcí na subjekty a jejich realizace.

[3] http://www.omgmarte.org

3. Platform-based Modelling
    - GRM (Generic Resource Modeling) - pro modelování společné platformy zdrojů na systémové úrovni a pro specifikaci jejich použití.
    - SRM (Software Resource Modeling) - pro modelování „multitask-based“ návrhu.
    - HRM (Hardware Resource Modeling) - pro modelování hardware platformy.
4. Model-based QoS Analysis
    - GQAM(Generic Quantitative Analysis Modeling) - pro anotace subjektů modelu, které jsou předmětem kvantitativní analýzy.
    - SAM(Schedulability Analysis Modeling) - pro anotace subjektů modelu, které jsou předmětem analýzy plánování.
    - PAM(Performance Analysis Modeling) - pro anotace subjektů modelu, které josu předmětem „performance analýzy“.

2.4 XML Metadata Interchange

XMI (XML Metadata Interchange) [14, 17] je OMG standard pro výměnu objektů modelu a metadat informací pomocí XML. Nejobecnější použití XMI je jako formát výměny pro diagramy UML mezi modelovacími nástroji a sklady metadat v distribuovaném prostředí, ačkoli může být také použit i pro jiné účely. XMI integruje tři klíčové standarty jako XML, MOF a UML. Specifikace definuje jak má být meta-model (který musí být založený na MOF) namapován na XML specifikaci. To znamená, že není pouze jeden XMI formát. Pro všechny verze UML je obvykle odlišný meta-model (založen na MOF) a proto je XMI specifikace pro všechny verze UML odlišná. XMI zachycuje model na třetí, druhé a první vrstvě MOF.



Obrázek 3: Vztah XMI a MOF

XMI specifikace definuje tyto pravidla:

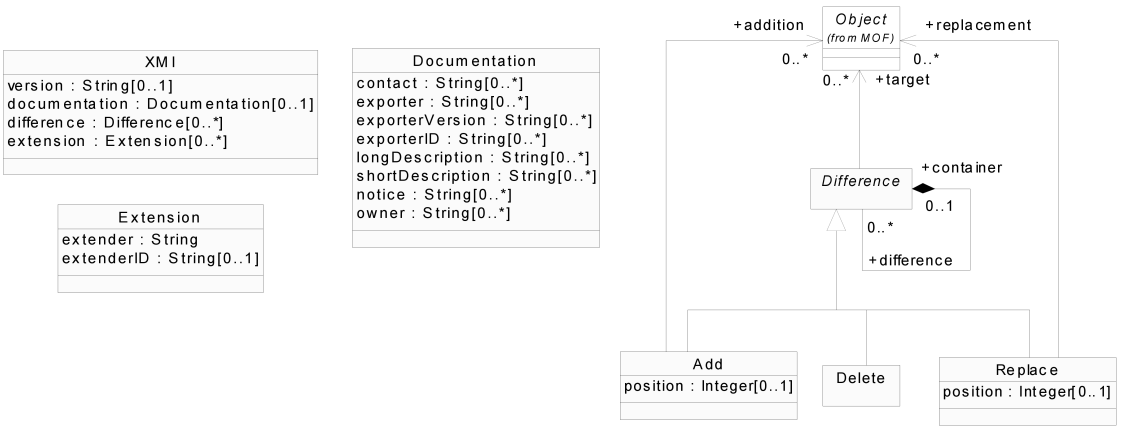
- pravidla pro generování XML Document Type Definitions (DTDs) z MOF based metamodels,

XMI	MOF	podpora schémat	zachycení změn modelu
1.1	1.3	-	-
1.2	1.4	-	-
1.3	1.4	x	-
2.0	1.4	x	x
2.1	2.0	x	x

Tabulka 1: Jednotlivé verze XMI ve vztahu k MOF

- pravidla pro generování XML dokumentů z MOF based metadata a MOF based metadata z XML dokumentů,
- pravidla pro konstrukci XMI související s DTD a XML daty,
- aktuální DTD podporující UML a MOF

Každý model je reprezentován ve schématu XML elementem, jehož jméno je název třídy, stejně tak i název complexType je název třídy. Deklarace typu jsou uvedeny jako vlastnosti třídy. Obsah modelů XML elementů odpovídají modelu tříd a nenařizují striktně pořadí vlastností. Jednotlivé třídy modelu jsou popsány ve specifikaci *Unified Modeling Language Infrastructure* [20] a všechny elementy diagramu UML pak ve specifikaci *Unified Modeling Language Superstructure* [21]. Verze 2.x byla radikálně přepracována vůči svým předchůdcům. Vyšší verze umožňuje mimo jiné zachycení změn modelu a přidání tzv. extensions. Pomocí extensions se mohou rozšířit třídy metamodelu. Některé CASE využívají extensions k uložení specifických vlastností daného nástroje jako např. pozice elementů diagramů, jejich barvu apod.



Obrázek 4: Class diagram struktury XMI a zachycení změn modelu [17]

2.5 Case nástroje

CASE (Computer-aided software engineering)[23] je označení pro počítačem podporované softwarové inženýrství. CASE nástroje jsou postaveny tak, aby podporovaly týmovou práci při vývoji systému, zajišťují sdílení rozpracovaných fragmentů, správu vývoje, sledují konzistenci modelu systému, automatizují některé procesy, hlídají dodržování zvolené metodiky, některé umožňují řízení celého životního cyklu aplikací. Primární možnosti CASE nástroje můžeme shrnout v těchto bodech:

- modelování IT systému pomocí diagramů (člověk lépe chápe obrázek než složitě psané slovo);
- generování zdrojového kódu z modelu (usnadňuje práci programátorům);
- zpětné vytvoření modelu podle existujícího zdrojového kódu (reverse engeneering);
- synchronizaci modelu a zdrojového kódu;
- vytvoření dokumentace z modelu;
- podporuje přenášení modelu mezi nástroji

Využití CASE nástrojů s sebou přináší celou řadu výhod, nejpodstatnější z nich jsou:

- vyšší produktivita práce;
- nižší chybovost;
- snazší údržba a další vývoj výsledného produktu;
- kvalitnější dokumentace;
- umožnění větší participace uživatelů na vývoji produktu;
- modelování chování budoucí aplikace

Existuje bohatá nabídka CASE nástrojů, jak komerčních, tak volně dostupných pro různé domény problémů a to včetně s podporou modelování embedded nebo real-time systémů. Přehledné rozdělení a popis CASE nástrojů nalezneme mimo jiné zde[4] [5].

[4] http://case-tools.org/

[5] http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

3 Procesně funkcionální jazyk e-$\mathcal{PFL}$

e-$\mathcal{PFL}$ je hybridní funkcionální jazyk určený pro modelování vestavných systémů v prvních fázích vývoje [4], který byl navržen Ing. Marekem Běhálkem, Ph.D. pro experimentální účely. Jeho konečná podoba vznikla postupnou evolucí z jazyků $\mathcal{PFL}$ a $\mathcal{PPFL}$. První zmiňovaný můžeme chápat jako jakýsi startovní experimentální jazyk disertační práce. Potřeba modelovat souběžně běžící funkční jednotky vedla k rozšíření konstrukce jazyka o definice paralelních úloh. Takto vznikl právě jazyk $\mathcal{PPFL}$. e-$\mathcal{PFL}$ je již cíleně určen pro tvorbu vestavných systémů a nese rysy obou zmiňovaných jazyků.

e-$\mathcal{PFL}$ pracuje s vysokou mírou abstrakce pro vývoj vestavěných systémů na nižší úrovni. Může být využit jako modelovací, nebo prototypový jazyk v rané fázi vývoje tak, aby došlo k případné eliminaci rizik při návrhu. Vychází z čistě funkcionální podmnožiny jazyka Haskell[6] a tu rozšiřuje o možnost použití proměnných. Použití proměnných je ovšem omezeno a z velké části řízeno kompilátorem. Díky tomu si e-$\mathcal{PFL}$ zachovává řadu dobrých vlastností funkcionálního programování a je i na jazykové úrovní blízký čistě funkcionálním jazykům. Představuje tak přehledný a snadno pochopitelný jazyk. Díky tomu jsme schopni například snadněji popsat stav vyvíjeného sytému (to může být v čistě funkcionálních jazycích poměrně obtížné), popsat vstupně výstupní operace nebo realizovat cykly.

3.1 Základní konstrukce jazyka

Vestavný systém je popsán jako soubor spolupracujících funkčních jednotek `Device`. Jde o běžný datotý typ definovaný v základním modulu `Prelude` (tento modul obsahuje základní knihovní funkce).

```
data Device = Process EmbProcess
            | Fair    [Device]
            | Unfair  [Device]
```

Definované jednotky jsou striktně složeny z vestavěných procesů `EmbProcess`. Tento typ není definován běžným způsobem a jeho podpora je vestavěna přímo do kompilátoru. Každá funkční jednotka je autonomní systém, který pracuje asynchronně a nezávisle na ostatních jednotkách. Tyto vestavné procesy řeší problémy spojené s komunikaci na koordinační vrstvě.

```
work :: a Int -> b Int ->(c Int, d Int)
work x y = (x, x+y)
```

Definice vestavěného procesu je rozšířena o proměnné. Pokud je návratovou hodnotou n-tice, musí být každý její element spojen s proměnnou. Proměnné reprezentují tak komunikační kanály a vestavěný proces konkrétní operaci s jasně definovaným vstupem a výstupem. Každá proměnná může být použita jako vstup právě jedné jednotky a také jako výstup právě jedné jednotky. Namodelované funkční jednotky jsou spuštěny pomocí nativní funkce `startDevice`.

```
startDevice :: Device -> EmbSystem -> [Annotation] -> ()
```

[6] http://www.haskell.org

Každá taková funkční jednotka je autonomní systém, který pracuje v principu asynchronně a nezávysle na ostatních jednotkách. Když je jednotka nastartována, snaží se vykonávat vestavné procesy, které obsahuje. V jedné chvíli může být vykonáván maximálně jeden proces. Vestavné procesy mezi sebou soutěží o to, který bude vykonán. Pokud aktuálně není vykonáván žádný vestavný proces, je vybrán nový kandidát a to na základě dostupnosti vstupu a *férovosti*. Při výběru musí být nejdříve splněna podmínka, že všechny vstupní komunikační kanály vestavného procesu obsahují hodnotu. Pokud tomu tak je, je vestavný proces schopen běhu. Mezi vestavnými procesy schopnými běhu je pak následně vybrán jeden. Jeho výběr je ovlivněn datovými konstruktory `Fair` (jsou vybírány elementy na nižší úrovni tak, aby každý potomek ve stromě měl stejnou možnost být vybrán) a `Unfair` (elementy na nižší úrovni stromu jsou vybírány podle pořadí v původní definici).

Vstupní hodnoty, reprezentované komunikačními kanály ztotožněnými s parametry vestavného procesu jsou procesem při jeho provedení *zkonzumovány*. Po provedení se vestavný proces snaží do výstupu, komunikačních kanálů spojených s návratovou hodnotou, zapsat vypočítané výsledky. Dokud se mu to nepovede, nemůže být výpočet ukončen a nemůže se začít s vykonáváním dalšího vestavného procesu. Hodnotu lze do výstupního kanálu zapsat ve chvíli, kdy neobsahuje žádnou hodnotu.

Jednou spuštěné funkční jednotky nelze zastavit ani modifikovat. Pro úplnost je nutné dodat, že vestavné procesy nelze používat běžným způsobem, nelze je tedy aplikovat přímo na nějaké konkrétní hodnoty. Funkční jednotky mohou být rozděleny do komponent `EmbComponent`.

```
data Mediator = Hume | MicroNET

data EmbSystem = EmbComponent [Character] Mediator | Emulator
```

Datový typ `EmbSystem` má dva datové konstruktory. Konstruktor `Emulator` slouží v první fázi vývoje, kdy ještě nechceme řešit rozdělení vyvíjeného vestavného systému do komponent. Druhý datový konstruktor je `EmbComponent`. Tento datový konstruktor má dva parametry. První umožňuje definovat jméno komponenty (jde o řetězec), druhý pak definuje použitého prostředníka.

Nakonec je možné změnit některé vlastnosti definovaných funkčních jednotek ve chvíli, kdy jsou spouštěny. Tyto změny můžeme provést prostřednictvím *annotací*. Použitím anotací jsme schopni změnit propojení systému na koordinační vrstvě, nastavení iniciálních hodnot nebo úroveň optimalizace cílového kódu. Anotace jsou definovány s použitím datového typu `Annotation`. Programátor nemusí používat tyto anotace přímo. Může používat například definované funkce. Příkladem může být funkce `rename`. Ta je v modulu `Prelude`. Lze ji využít k přejmenování některého ze vstupů či výstupů. Takto lze změnit propojení jednotlivých jednotek.

```
data Attribute  = CAttribute  [Character] [Character]
data Annotation = CAnnotation [Character] [Attribute]

rename::[Character]->[Character]->Annotation
rename x y = CAnnotation "rename"
               [(CAttribute "old" y),
```

```
(CAttribute "new" x)]
```

Definované a spuštěné funkční jednotky komunikují prostřednictvím definovaných komunikačních kanálů. Každý takový komunikační kanál ve své podstatě spojuj právě dva zdroje. Jak již bylo uvedeno, konkrétní komunikační kanál může sloužit jako vstup maximálně jedné jednotce a jako výstup také maximálně jedné jednotce. Pokud není spojen s nějakou funkční jednotkou, může být použit jako výstup například do nějakého síťového proudu, nebo například na standardní výstup.

3.2 Praktická hlediska

Z praktického hlediska je k dispozici e-*PFL* překladač[7], který pracuje nad definovanou gramatikou. Překladač je schopen vygenerovat cílový kód v jazyce C#. K tomuto vygenerovanému kódu je pak připojeno jednoduché běhové prostředí a tento vysledek lze zkompilovat překladačem jazyka C# a spustit. Nastavení vlastností komunikace může programátor ovlivnit v rámci konfigurace systému změnami v XML konfiguračním souboru. Uvedený překladač a podpůrné nástroje byly vytvořeny v programovacím jazyce C# na platformě .NET Micro Framework[8].

[7] Nástroje e-*PFL* a jejich popis: http://www.cs.vsb.cz/behalek/epfl
[8] http://www.microsoft.com/en-us/netmf

4 Teoretická východiska a nástin řešení problematiky

4.1 Průzkum technologie UML pro popis vestavěných systémů

V prvním kroku bylo nutné nastudovat možnosti technologie UML pro popis vestavěných systémů. Detaily k teorii jsou k dispozici v předešlé kapitole *Související technologie a přístupy*. Diplomová práce není primárně zaměřena na vestavné systémy jako takové, proto se tomuto tématu nevěnuji do hloubky. UML je nástroj, který je určen pro popis a analýzu objektově orientovaných, nebo tomuto paradigmatu příbuzným systémům. Na druhou stranu poskytuje uživateli určitou volnost v použití, interpretaci a možnosti rozšíření tak, že může být použit i pro jiné domény, než je striktně OOP. Je tedy částečně na uživateli, jakým způsobem využije UML a při kterých fázích vývoje vestavných systému ho bude používat. Může ho použít kupříkladu pouze pro popis kritických částí systému, jako neformální diagramy některých částí, které jsou právě diskutovány, nebo pro komplexní popis veškeré budoucí funkcionality systému a to jak ze statických, tak i dynamických aspektů. Pro specifické účely vestavných systémů si můžeme vystačit s dostupnými možnostmi UML spolu s *Lightweight* rozšířením sémantiky, jako je zavedení nových stereotypů (stereotypes), označkování hodnot (tagged values) a omezení (constraints). Existuje také celá řada UML profilů pro specifické požadavky vestavěných resp. Real-Time systémů jako je např. profil MARTE.

Z hlediska této práce nebylo objektem bádání přímá specialize a využití UML pro SW/HW část skutečných vestavěných systémů, ale je víceméně orientována k možnostem využití UML pro generování modelů z překladače/simulátoru e-*PFL*, který je sám o sobě určený pro modelování vestavných systémů v prvních fázích vývoje. Zde si vystačíme se základní sadou elementů, které nabízí UML 2.0 a vyšší.

4.2 Možnosti práce s UML diagramy, vhodný nástroj a formát uložení

Další krok byl zaměřen na vhodný formát pro uložení UML modelů, přenos do CASE nástroje, následné zpracování a vizualizaci diagramů. Napřed byly prozkoumány tyto CASE nástroje a jejich možnosti: *Sparx Systems Enterprise Architect (trial)*, *Altanova UModel 2011 (trial)*, *IBM Rational Software Architect 8.0 (plná verze, univerzitní licence)*, *ArgoUML (open source)* [9], *Papyrus (open source)* [10], *Modelio (open source)* [11]. Po průzkumu činnosti těchto nástrojů se došlo z hlediska formátu uložení modelu k těmto závěrům:

1. CASE nástroj umožňuje uložit model a diagramy ve výrobcem specifickém formátu souboru, který je často vnitřně reprezentován pomocí XML. Zaměřit se na tento formát by nebylo příliž vhodné, jelikož dokumentace významu struktury souboru není obvykle oficiálně dostupná a znamenalo by to v praxi pouze vypozorovat, co daný nástroj ukládá při změnách modelu v editoru, což má jistě velká omezení. Navíc je toto řešení úzce spjaté pouze s jedním nástrojem a není tak zaručena interkompatibilita s jinými.

[9] http://argouml.tigris.org/
[10] http://www.papyrusuml.org/
[11] http://www.modelio.org/

2. Některé z nástrojů také nabízí alternativní formáty souborů pro import jako CSV, EMF ECORE, nebo rovnou import specifických souborů jiných výrobců, jako např. *.mdl od Rational Rose apod. Zaměřit se na jeden z těchto formátů by mělo stejné nevýhody jako v předchozím bodě.

3. Valná část nástrojů podporuje export, resp. import modelu ze souborového formátu XMI. Některé jako *Modelio*, *ArgoUML* dokonce nemají vlastní formát uložení a používají přímo jen XMI. I když tento formát nezahrnuje všechny potencionální možnosti nástroje, jak je tomu u CASE-specifického formátu, pro účely přenosu modelů a následné vytvoření diagramů je zcela vyhovující. Jak je popsáno v kapitole 2.4, jde o standartizovaný způsob uložení modelů spravovaný konsorciem OMG a hojně využívaným v CASE nástrojích. Jedná se o generické řešení, které zaručí úpnou, nebo alespoň částečnou kompatibilitu s většinou CASE nástrojů.

Zásadní rozdíl je v podporovaných verzích XMI a UML. Mezi verzi XMI 1.x a 2.x jsou značné rozdíly. Velká část zkoumaných CASE již pracuje se zmiňovanou novější verzi (kromě *ArgoUML*), tedy XMI 2.1 a verzi UML minimálně 2.1.1. Bylo tedy rozhodnuto, že veškeré úsilí bude zaměřeno na kombinaci XMI 2.1, UML 2.1.1 a MOF 2.0, ačkoli jsou již k dispozici novější specifikace, ale ty jsou implementované zatím jen sporadicky, pouze v nejnovějších verzích CASE nástrojů.

Při provádění prvního experimentu, založeném na vytvoření testovací aplikace, která v prvních fázích vygeneruje validní XMI soubor, obsahující libovolné třídní diagramy se základními prvky (např. atributy, metody, parametry metod, vychozí hodnoty parametrů, vazby, generalizace, závislosti apod.), byly zjištěny problémy s interkompatibilitou XMI v CASE nástrojích. Nebo-li pokud jeden nástroj načetl soubor a označil jej za validní, tak druhý měl například potíže s interpretací vazeb mezi třídami apod. To vedlo také k následným pokusům o export testovacích UML diagramů z jednoho CASE nástroje a import do druhého, které dle původních předpokladů né vždy proběhly v pořádku.

Přestože OMG poměrně striktně specifikuje MOF a mapování MOF na XML (nebo-li XMI), tak jednotlivé nástroje mají stále poměrně problémy s porozuměním modelu, který byl uložen v jiném nástroji. To také dokazují množící se dotazy v sekcích FAQ na stránkách výrobců CASE nástrojů. Důvodem je jednak často nesprávná implementace standardů, ale také v současné době různorodost verzí UML, MOF, XMI a v jejich vzájemné kombinaci. Další problém je, že se často zneužívá XMI tag „extension“, určený pro uložení specifických informací pro daný CASE nástroj, jako je např. vzhled a pozice grafických elementů výsledného diagramu. Vedle těchto doplňkových informací si tam pak výrobci SW někdy přibalí informace týkající se samotného modelu, jako vazba mezi elementy, definice stereotypů apod. Potom nástroj druhého výrobce nedokáže načíst všechny potřebné informace z takového souboru. Není proto výjimkou, že některé CASE nástroje obsahují vnitřní implementace, které zvyšují interkompatibilitu generovaných souborů jinými výrobci. To je realizováno buď poloautomaticky tak, že si uživatel zvolí při importu název a verzi nástroje v kterém byl soubor původně vytvořen, nebo to automaticky rozezná ze sekce *Documentation* dle atributů *exporter* a *exporterVersion*. Konsorcium OMG si je vědoma problémů s kompatibilitou výměny dat modelu mezi nástroji a vytvořilo

Wiki věnující se tomuto tématu [12]. Tam jsou také k dipozici testovací XMI soubory, které obsahují všechny potencionální možnosti tak, aby prověřily, zda CASE nástroj má standard implementovaný dostatečně a správně, či nikoli. V historii dokonce vznikly i služby na transformaci XMI souborů pro konkrétní verze CASE nástrojů, jako např. XMI2XMI Transformation[13].

Nejenom díky problémům s interkompatibilitou jsem se rozhodl zaměřit při praktické realizaci této práce na jeden primární CASE nástroj, a to *IBM Rational Software Architect 8.0*, který poskytuje dostatečný komfort a disponuje bohatými možnostmi při práci s modely a UML diagramy. Zároveň bylo možné využívat plnou komerční verzi díky univerzitní licenci. S RSA se pracuje velmi intuitivně. Po naimportování modelu ze souboru XMI se celá struktura elementů zobrazí v *Project explorer* a pak již lze libovoně vytvářet diagramy z jednotlivých elementů modelu. Na výběr jsou předdefinované šablony diagramů jako *Class Diagram*, *Use Case Diagram*, *Activity Diagram* atd., ale také *Topic Diagram* a *Freedom Diagram* pro volnější tvorbu a přizpůsobení diagramů „na míru".

Předmětem průzkumu bylo také zjistit, zda již existuje nějaká opensource knihovna, která umožňuje manipulaci s UML modely a uložení do XMI. Většina komplexních knihoven je implementována v jazyce Java. Asi mezi nejznámější patří *Eclipse Modeling Framework Project (EMF)* [14]. Také nejvíce informací k XMI je spojeno s jazykem Java a Eclipse, jelikož se na specifikaci podílí konsorcium společností, které podporují a preferují hlavně tyto platformy (např. IBM). Pro platformu .NET (která je objektem zájmu této DP), jsou k nalezení spíše kratší ukázky kódu na blozích a web stránkách určeným developerům.

4.3 Výběr vhodných části e-$\mathcal{PFL}$ modelu pro generování diagramů

V další fázi bylo nutné nastudovat procesně funkcionální jazyk e-$\mathcal{PFL}$, včetně jeho překladače a vybrat jeho vhodné části, které bude přínosné zobrazit jako diagramy.

Při zhostění se problému vizualizace UML diagramu z e-$\mathcal{PFL}$ je největší problém to, že jazyk vychází z čistě funkcionálního jazyka HASKELL. Jazyky patřící do skupiny funkcionálních, přistupují k programu jako k matematickému výrazu resp. funkci. Provádění programu spočívá v deterministickém vyhodnocování výrazů (funkcí) s aplikovanými argumenty, který vede v jediný výsledek. V podstatě nabízí jen funkce, které mají nějáký vstup a výstup, ale žádný stav. Mají pouze pravidla na mapování vstupu na výstup, bez chování. Tyto jazyky vycházejí z teorie funkcí lambda-kalkul. Z tohoto důvodu se obecně UML (které vychází původně z OOP paradigmatu) jen velmi těžce adaptuje na popis funkcionálních jazyků.

Na druhou stranu nás v podstatě nemusí zajímat přesné zachycení a vyhodnocení samotných funkcí a jejich popis pomocí UML diagramů, ale zajimavější je pro konstruktéra vestavěných systémů názorně vidět, statické i dynamické aspekty samotného simulovaného vestavěného sytému popsaného v e-$\mathcal{PFL}$. Zainteresované osoby mohou případně za

[12] http://www.omgwiki.org/model-interchange/
[13] http://modeling-languages.com:8080/mdeOnlineServices/xmiinput
[14] http://www.eclipse.org/modeling/emf/?project=emf

přispění takových diagramů lépe pochopit a verifikovat navrhovaný systém již v ranné fázi vývoje.

Jak již bylo napsáno v kapitole 3, každý vestavěný systém simulovaný v e-*PFL* se skládá z komponent (components) a funkčních jednotek (devices), které mají své vstupy (inputs) a výstupy (outputs). Jednotlivé jednotky mezi sebou vzájemě komunikují tak, že výstup jedné jednotky může být vstupem druhé. Z tohoto hlediska je velice užitečné generovat diagram, který nám dává představu o celkovém designu systému, tj. z jakých komponent, jednotek, vstupů a výstupů se skládá a jak jsou mezi sebou jednotlivé části propojeny a jak komunikují. Zde se nám nabízí třídní diagram, který je pro popis statického náhledu nad těmito prvky systému, včetně příslušných vazeb vhodný. Jednotlivé třídy pak dědí vlastnosti z bázových tříd *Embedded component*, *Embedded device* a *Device I/O*. Agregační vazby mezi konkrétní jednotkou a vstupem, nebo výstupem, mají v popisu názvy procesů, které komunikují a manipulují s jednotlivými proměnnými (vstupy a výstupy). Další užitečnou vazbou je *závislost* mezi vstupy a výstupy jednotek (který výstup jedné jednotky je vstupem druhé jednotky). Takový diagram nazveme *Overview diagram*.

Při průzkumu UML se našla jistá analogie mezi prvky deployment diagramu a prvky reprezentovanými v e-*PFL*. Tento diagram poskytuje element typu *Component* a *Device*, což nám přinese lepší přehlednost, ale i částečné zjednodušení, jelikož odpadnou potřebné bázové třídy, jak je tomu u předešlého diagramu. Samotné vstupy a výstupy budou reprezentovány třídou a vazby budou řešeny stejně, jak je tomu u *Overview diagram*. Tento diagram je v podstatě jinou reprezentací předešlého návrhovaného diagramu.

Podíváme-li se blíže na činnost e-*PFL*, tak nástroj vykonává chronologicky tyto fáze při překladu:

1. syntax parsing
2. device checking
3. type checking
4. static analyzing
5. (configuration)
6. target code generation
7. (simulation)

U e-*PFL* nás z hlediska vizualizace UML diagramů zajímají dvě fáze a to konfigurační a simulační. e-*PFL* nabízí tzv. konfigurační běh, neboli spuštění modelovaného systému v určité výchozí konfiguraci, kdy se při startu například nastaví hodnoty vstupů a výstupů. Z praktického hlediska je velmi užitečné vygenerovat tzv. diagram konfigurace. V tomto případě nás zajímají zejména konkrétní hodnoty, které jsou na vstupech a výstupech, jejich datové typy a zobrazení původních názvů vstupů a výstupů v případě použití anotací pro jejich přejmenování. Překladač využívá XML konfigurační soubor *[jmeno souboru].conf.xml*

pro nastavení parametrů konfiguračního běhu. Diagram nazvaný *Configuration diagram* bude realizovaný pomocí instancí tříd (neboli objektů). K vytvoření těchto instancí potřebujeme konkrétní třídy, z kterých se vlastní instance budou realizovat. Proto musí být součástí tohoto diagramu také tzv. bázový model konfigurace (*Base configuration diagram*).

Při simulační fázi je hlavním objektem zájmu zachycení komunikace jednotlivých prvků systému v čase, změny jejich stavů a případně volání interních funkcí. Základní diagramy, které mohou být pro tyto účely využity jsou *stavový diagram*, *sekvenční diagram* a *diagram aktivit*. Jednotlivé jednotky (devices) běží ve vlastních vláknech a mohou pracovat mezi sebou konkurenčně (fair a unfair režim). Takový režim není jednoduchý pro zachycení pomocí diagramů, jelikož se při generaci musíme postarat o synchronizaci na nížší úrovni. Pro vývojáře může být také zajímavé zachytit dynamické aspekty jedné jednotky, nebo jednoho vstupu a výstupu. Může jít o aktualizaci, nebo konzumaci kokrétních hodnot jednotlivých vestavěných proměnných, nebo přenesení hodnot do datového proudu. Pokud by jsme chtěli diagram, který popíše systém v širším kontextu, můžeme například využít diagram aktivit, který umožňuje také zachycení nezávislého paralelního běhu. Další problém při budování diagramu je, že samotné jednotky obsažené v komponentách pracují v nekonečném cyklu a provádějí akce, dokud např.uživatel nezastaví program stiskem klávesy. Otázkou tedy je, po jaké opakující se sekvenci kroků se samotný diagram považuje za kompletní. Zde si můžeme zavést pojem tzv. *počet strojových cyklů*. Znamená to, kolik cyklů jednotka provede. Technicky je cyklus v e-*PFL* řešen pomocí cyklu *while* v hlavní spouštěcí metodě vlákna. Tento parameter musí být předem nastaven uživatem. Přitom hodnota nemusí být často vysoká. Většinou stačí generovat 5-10 cyklů, aby architekt vestavného systému viděl, jaké hodnoty se mění na vstupech a výstupech, zda jsou korektní a jaké procesy při tomto vstupují do hry.

UML diagramy a jejich použití:

- **Stavový diagram** - zachycuje stavy objektu, přechody mezi nimi a události. Stavem můžeme označit situaci, kdy objekt čeká na událost, nebo se nějákým způsobem chová. Ten může obsahovat libovolný počet akcí a aktivit. Stav je v daném okamžiku určen hodnotami atributů daného objektu, aktuálně vykonanou aktivitou a relací s objekty. Každý diagram stavů zachycuje v daný okamžik změny stavu pouze jednoho objektu. Z tohoto hlediska se v naší aplikaci hodí pro detailnější zachycení konkrétní části vestavěného systému, který mění své stavy. Konkrétní části se myslí např. Device, Process, EmbeddedVariable. Můžemé také zachytit změny na vstupech a výstupech konkrétní jednotky. Události vyvolávající přechody pak můžou být vestavěné procesy.

- **Diagram aktivit** - tento diagram se používá k zachycení procesů, workflow, nebo procedurální logiky (vykonávání jednotlivých funkcí). Znázorňuje tok řízení z aktivity do aktivity. V porovnáním se stavovým diagramem jsou si velmi podobné, oba ukazuji sekvenci stavů a podmínky způsobující tyto přechody. Hlavní rozdíl je v tom, že u diagram aktivit se stav může týkat více objektů najednou. Není striktně dáno, že každý stav může náležet pouze jednomu objektu, jak je tomu u

stavového diagramu. Výhodou je pak i to, že umožňuje paralelní vykonávaní akcí dvou nezávislých řídících toků. Tato vlastnost může být v našem případě výhodná, protože jednotlivé jednotky navrhovaného systému vykonávají činnost paralelně v oddělených vláknech.

- **Sekvenční diagram** - zachycuje zprávy, které si mohou objekty mezi sebou synchronně, nebo asynchronně posílat v časové posloupnosti. Nejdůležitějším prvkem je tzv. čára života, která reprezentuje jednotlivé účasníky interakce a dále životnost objektu samotného. Čára života může indikovat, zda objekt existoval před posláńím první zprávy a zda bude jeho životnost ukončena na konci sekvence. Sekvenční diagram umožňuje v notaci zahrnout i stavy objektů. Tento diagram by mohl být aplikovaný na konkrétní vestavnou jednotku a popsat sekvenci volání jednotlivých funkcí, které jsou na objektech typu Device, Process, EmbeddedVariable. Zde můžeme zachytit např. nastavení, nebo konzumaci hodnot vestavěné proměnné, volání anotací na jednotce apod.

5 Návrh a realizace vlastního nástroje

5.1 Návrh nástroje

Při návrhu nástroje pro generování UML diagramů bylo bráno v potaz to, že překladač e-$\mathcal{PFL}$ je již vytvořen pro platformu .NET. Z tohoto důvodu bylo logicky rozhodnuto o vytvoření vlastní obecné assembly pro platformu Microsoft .NET 3.5 v jazyce C#, která bude posléze referencována e-$\mathcal{PFL}$ a využívat její možnosti. Assembly pojmenovaná *XMILibrary.dll* je napsána obecně tak, aby ji šlo použít i pro jiné projekty. Při průběžném testování nové knihovny vznikl také projekt *TestXMI*. Jedná se o konzolovou testovací aplikaci, která generuje jakési testovací modely pro otestování funkčnosti *XMILibrary*.

Při samotném programovaní knihovny nebyl striktně nenásledován objektový model, který je popsán v OMG specifikacích [20, 21], ale byl pro naše účely značně zjednodušen. Původní model je velmi komplexní, čítá okolo 300 tříd a většina UML elementů disponuje vlastnostmi, které při našem vytváření diagramů nepoužijeme. Nebylo by ani v silách jednoho řešitele DP vytvořit knihovnu, obsahující všechny prvky a vlastnosti poskytující UML 2.1.

V prvním kroku bylo nutné navrhnout efektivní podporu serializace objektů tak, aby mohla být s úspěchem a jednoduše použita pro všechny elementy XMI modelu (i v případě budoucího rozšíření). Tyto elementy jsou v projektu reprezentovány samostatnými třídami. Servis těmto třídám zajišťuje třída *Serializer*, která se stará o samotnou serializaci do XMI. Jelikož každé XMI obsahuje více jmenných prostorů, byla s výhodou použita pro správu XML a vytváření XML elementů .NET assembly *System.Xml.Linq*. Ta nám oproti klasickým přístupům značně zjednodušuje práci s konstrukcí strukturovaných XML dokumentů. Každá třída, která reprezentuje UML element, implementuje rozhraní *ISerializableXMI* prostřednictvím bázové třídy *Element*. Pak máme k dispozici v každém konkrétním elementu override metody *SerializeAttributes* a *SerializeCompositions*. První ze zmiňovaných slouží k serializaci takových vlastností, které jsou v XMI prezentovány jako atributy elementů. V druhém případě se jedná o serializaci složených konstrukcí, jako například kolekce *Operation* v případě elementu *Class*.

```
public interface ISerializableXMI
{
    void SerializeAttributes ( Serializer   serializer ,  XElement currentXElement);

    void SerializeCompositions(Serializer  serializer ,  XElement currentXElement);
}
```

Výpis 1: Rozhraní *ISerializableXMI*

Bázová třída nám dále poskytuje atributy *ID*, *UUID* (Unique identificator) a kolekci komentářů *ownedComment*, které jsou dle specifikace XMI obsaženy v každém elementu. Každý element reprezentovaný třídou dále implementuje různé rozhraní podle podporovaných vlastností např. pojmenování elementu - *INamedElement*, nastavení úrovně viditelnosti - *IVisibilityElement*, element může být použit v balíčku - *IPackageableElement* atd.

XMILibrary nabízí v současné době podporu pro tyto UML elementy (podrobný popis významu je uveden v [20, 21]):

- **namespace XMILibrary.Base** - Association, Class, Comment, DefaultValue, Dependency, Generalization, InstanceSpecification, InstanceValue, Interface, InterfaceRealization, Model, Operation, Package, Parameter, Profile, Property, Slot
- **namespace XMILibrary.Activities** - Activity, ActivityAction, ActivityFinalNode, ControlFlow, DecisionNode, FlowFinalNode, ForkNode, InitialNode, JoinNode, MergeNode, ObjectFlow, ObjectNode
- **namespace XMILibrary.Deployment** - Artifact, Component, Device, ExecutionEnvironment, Node
- **namespace XMILibrary.UseCase** - Actor, UseCase, Extend, Include

Při praktické realizaci byly použité tyto technické prostředky:

- Kompletní zdrojové kódy e-*PFL*
- Microsoft Visual Studio Professional 2010
- IBM Rational Software Architect 8.0 - CASE nástroj pro modelování v UML
- StyleCop (VS add-in) - analyzuje zdrojový kód z hlediska „čistoty“ psaní zdrojového kódu, hlídá formátování a přehlednost kódu C#, obsahuje několikset pravidel a stylů pro přehledné a obecně uznávané zásady kladené na moderně psaný zdrojový kód.
- GhostDoc (VS add-in) - dokumentování zdrojového kódu, automatické generování dokumentace.

5.2 Příklady konstrukce elementů UML modelů

Tato kapitola demonstruje příklady použití vlastní knihovny *XMILibrary*. Demonstrativní příklady samozřejmě nepostihují všechny kombinace a možnosti této knihovny. Uložené XMI soubory byly následně naimportovány a zpracovány pomocí *IBM Rational Software Architect 8.0* v konkrétní diagramy. Obsah vygenerovaných XMI souborů se nachází kvůli rozsahu v příloze B. Další komplexní příklady jsou pak k nalezení na přiloženém CD.

```
// Vytvoreni testovaciho modelu
XMI xmi = new XMI("TestModel");

// Konstrukce hlavnich elemetu
Interface iterface1 = new Interface("Inteface1");
Class class1 = new Class("Class1", true);
Class class2 = new Class("Class2");
Class class3 = new Class("Class3");
```

```
// pridani atributu
iterface1 .OwnedAttribute.Add(new Property("Property1", DataTypeKind.integer));
class1.OwnedAttribute.Add(new Property("Property1", DataTypeKind.integer));
class2.OwnedAttribute.Add(new Property("Property2", DataTypeKind.boolean));

// pridani operaci
Operation operation1 = new Operation("Operation1",
   new Parameter("param1", ParameterDirectionKind.@in, DataTypeKind.@string),
   new Parameter("param2", ParameterDirectionKind.@return, DataTypeKind.@string));
class1.OwnedOperation.Add(operation1);

// Pridani poznamky k elementu Class1
class1.AddComment("Abstract class");

// Konstrukce realizace Interface (Class1 ---> Iterface1)
class1.AddInterfaceRealization(iterface1);

// Konstrukce generalizace (Class2 ---> Class1)
class2.AddGeneralization(class1);

// Konstrukce composite association (Class3 <>---> Class2)
Association association = class3.AddAssociation(class2, "association");
association.MemberEnd[0].Aggregation = AggregationKind.composite;
association.MemberEnd[0].Lower = 1;
association.MemberEnd[0].Upper = UnlimitedNatural.Parse("2");

// Pridani vsech elementu do modelu
xmi.Model.OwnedMember.Add(class3);
xmi.Model.OwnedMember.Add(class2);
xmi.Model.OwnedMember.Add(class1);
xmi.Model.OwnedMember.Add(iterface1);

// Ulozeni vystupu do XMI souboru
xmi.Save(fileName);
```

Výpis 2: Příklad konstrukce tříd, rozhraní, asociace, generializace, realizace

```
// Vytvoreni testovaciho modelu
XMI xmi = new XMI("TestModel");

Activity activity = new Activity("Activity diagram");

InitialNode init = new InitialNode("Start");
activity .Node.Add(init);

ActivityAction action1 = new ActivityAction("Action1");
activity .Edge.Add(init.AddControlFlow(action1));
activity .Node.Add(action1);

DecisionNode decision = new DecisionNode("x > 10");
activity .Edge.Add(action1.AddControlFlow(decision));
activity .Node.Add(decision);
```

```
ActivityAction action2 = new ActivityAction("Action2");
activity.Edge.Add(decision.AddControlFlow(action2,"false"));
activity.Node.Add(action2);

ForkNode fork = new ForkNode("Fork");
activity.Edge.Add(action2.AddControlFlow(fork));
activity.Node.Add(fork);

ActivityAction action3 = new ActivityAction("Action3");
activity.Edge.Add(fork.AddControlFlow(action3));
activity.Node.Add(action3);

ActivityAction action4 = new ActivityAction("Action4");
activity.Edge.Add(fork.AddControlFlow(action4));
activity.Node.Add(action4);

JoinNode join = new JoinNode("Join");
activity.Edge.Add(action3.AddControlFlow(join));
activity.Edge.Add(action4.AddControlFlow(join));
activity.Node.Add(join);

ActivityAction action5 = new ActivityAction("Action5");
activity.Edge.Add(join.AddControlFlow(action5));
activity.Node.Add(action5);

activity.Edge.Add(decision.AddControlFlow(action5, "true"));

ActivityFinalNode activityFinalNode = new ActivityFinalNode("Exit");
activity.Edge.Add(action5.AddControlFlow(activityFinalNode));
activity.Node.Add(activityFinalNode);

xmi.Model.OwnedMember.Add(activity);

// Ulozeni vystupu do XMI souboru
xmi.Save(fileName);
```

Výpis 3: Příklad konstrukce diagramu aktivit

```
// Vytvoreni testovaciho modelu
XMI xmi = new XMI("TestModel");

// Konstrukce hlavnich elemetu
Artifact artifact = new Artifact("Artifact1");
Component component = new Component("Component1");
Device device = new Device("Device1");
ExecutionEnvironment executionEnvironment = new ExecutionEnvironment("
    ExecutionEnvironment1");
Node node = new Node("Node1");

// Konstrukce zavislosti a asociaci
component.AddDependency(device);
device.AddDependency(node);
```

```
component.AddAssociation(executionEnvironment, string.Empty);
component.AddDependency(device);

// Properties & operations
component.OwnedAttribute.Add(new Property("Property1"));
component.OwnedOperation.Add(new Operation("Operation1"));
device.OwnedAttribute.Add(new Property("Property1", DataTypeKind.integer));

// Konstrukce zavislosti a asociaci
component.AddDependency(device);
device.AddDependency(node);
component.AddAssociation(executionEnvironment, string.Empty);
executionEnvironment.AddDependency(artifact);

// Pridani vsech elementu do modelu
xmi.Model.OwnedMember.Add(artifact);
xmi.Model.OwnedMember.Add(component);
xmi.Model.OwnedMember.Add(device);
xmi.Model.OwnedMember.Add(executionEnvironment);
xmi.Model.OwnedMember.Add(node);

// Ulozeni vystupu do XMI souboru
xmi.Save(fileName);
```

Výpis 4: Příklad konstrukce diagramu nasazení

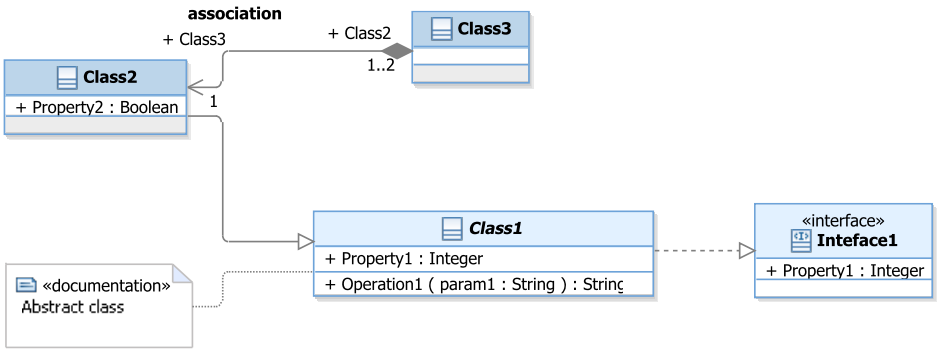
```
// Vytvoreni testovaciho modelu
XMI xmi = new XMI("TestModel");

UseCase useCase1 = new UseCase("Use case 1");
UseCase useCase2 = new UseCase("Use case 2");
UseCase useCase3 = new UseCase("Use case 3");
UseCase useCase4 = new UseCase("Use case 4");
UseCase useCase5 = new UseCase("Use case 5");
Actor actor1 = new Actor("Actor1");
Actor actor2 = new Actor("Actor2");

// Konstrukce generalizace, extension, include, asociace
actor2.AddGeneralization(actor1);
useCase1.AddExtension(useCase2);
useCase3.AddInclude(useCase4);
useCase4.AddGeneralization(useCase5);
actor1.AddAssociation(useCase2, string.Empty);
actor1.AddAssociation(useCase5, string.Empty);

// Pridani vsech elementu do modelu
xmi.Model.OwnedMember.Add(useCase1);
xmi.Model.OwnedMember.Add(useCase2);
xmi.Model.OwnedMember.Add(useCase3);
xmi.Model.OwnedMember.Add(useCase4);
xmi.Model.OwnedMember.Add(useCase5);
xmi.Model.OwnedMember.Add(actor1);
xmi.Model.OwnedMember.Add(actor2);
```

```
// Ulozeni vystupu do XMI souboru
xmi.Save(fileName);
```

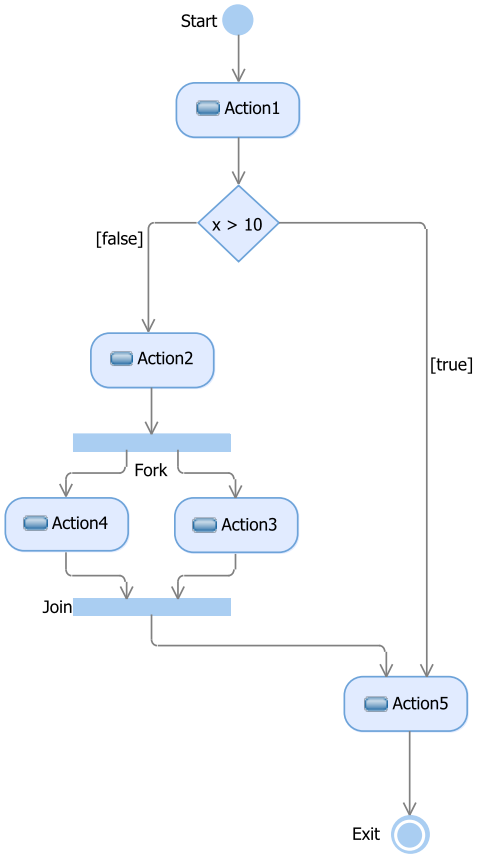
Výpis 5: Příklad konstrukce diagramu případů užití

```
// Vytvoreni testovaciho modelu
XMI xmi = new XMI("TestModel");

// Konstrukce trid
Class class1 = new Class("Class1");
Property property1 = new Property("Property1", DataTypeKind.@string);
class1.OwnedAttribute.Add(property1);
Class class2 = new Class("Class2");
Property property2 = new Property("Property2", DataTypeKind.@string);
class2.OwnedAttribute.Add(property2);

// zavislost trid
class1.AddDependency(class2);

// konstrukce instanci trid (objektu)
InstanceSpecification class1Instance = new InstanceSpecification("Object1", class1);
class1Instance.Slot.Add(new Slot(property1, "concrete value 1"));
InstanceSpecification class2Instance = new InstanceSpecification("Object2", class2);
class2Instance.Slot.Add(new Slot(property2, "concrete value 2"));

// zavislost objektu
class1Instance.AddDependency(class2Instance);

// Pridani vsech elementu do modelu
xmi.Model.OwnedMember.Add(class1);
xmi.Model.OwnedMember.Add(class2);
xmi.Model.OwnedMember.Add(class1Instance);
xmi.Model.OwnedMember.Add(class2Instance);

// Ulozeni vystupu do XMI souboru
xmi.Save(fileName);
```

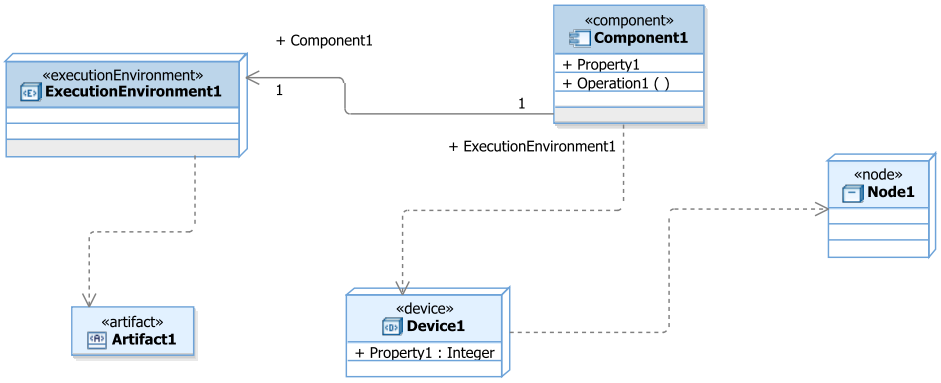
Výpis 6: Příklad konstrukce diagramu objektů



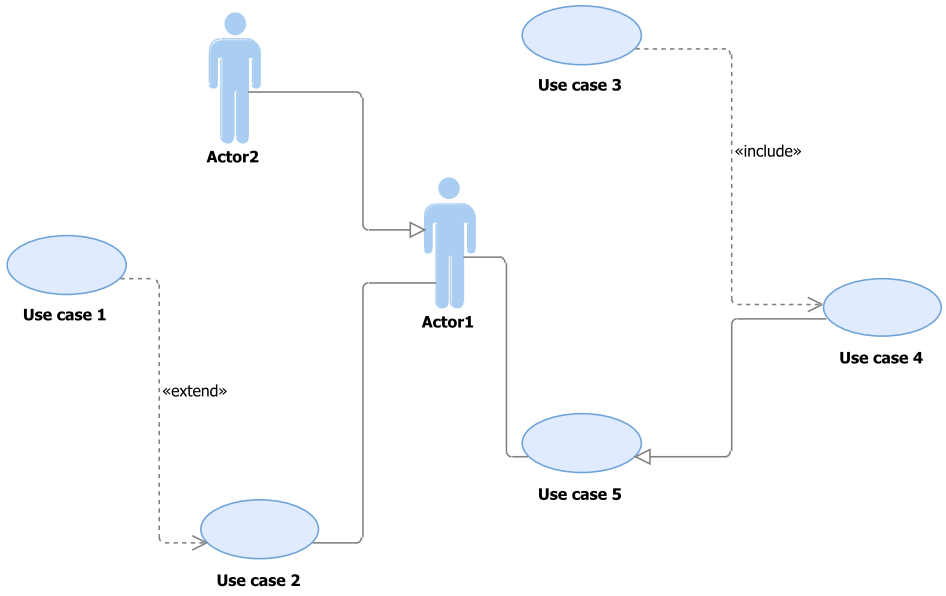
Obrázek 5: UML diagram generovaný z výpisu č.2



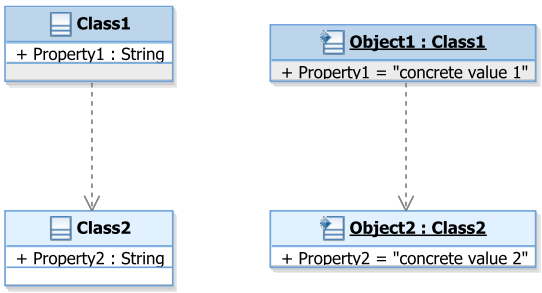
Obrázek 6: UML diagram generovaný z výpisu č.3



Obrázek 7: UML diagram generovaný z výpisu č.4



Obrázek 8: UML diagram generovaný z výpisu č.5



Obrázek 9: UML diagram generovaný z výpisu č.6

5.3 Integrace s e-*PFL*

Hlavní aplikace e-PFL.exe je konzolová aplikace, zajišťující překlad popsaného vestavěného systému v e-*PFL* do zdrojového kódu v C# a případně také následně automaticky zkompilovat do spustitelného souboru typu exe. Výsledný soubor již představuje samotný vestavěný systém a po jeho spuštění lze simulovat jeho realný běh. Vygenerované zdrojové kódy v C# můžeme rovněž připojit k debuggeru a ladit tak libovolně jednotlivé části systému.

Po průzkumu možné integrace vlastní knihovny XMILibrary do e-*PFL* byly použity dva oddělené přístupy:

1. Pro diagramy popisující aktuální konfiguraci a celkovou strukturu systému je nejlepším výchozím bodem samotná aplikace e-PFL.exe. Zde je v době překladu známá konfigurace a struktura popisovaného vestavěného sytému. Aplikace obsahuje konfigurační třídy, z kterých lze poměrně snadno získat informace k vytvoření srozumitelných diagramů.

2. Pro diagramy popisující dynamické aspekty systému je nutné, aby systém během generace prováděl vlastní běh, procházel konkrétní části systému, volal jednotlivé funkce, manipuloval se vstupy a výstupy apod. Takovou podporu diagramů lze automaticky integrovat do výsledného generovaného kódu C#, který zajišťuje e-PFL nástroj.

5.3.1 Integrace generování diagramů popisující pohled na aktuální konfiguraci a celkovou strukturu systému

Při integraci generování XMI diagramů byly původní zdrojové kódy e-*PFL* rozšířeny o tyto třídy:

- **XMIDiagramGenerator** - třída se stará o generování jednotlivých modelů na základě dat získaných z konfiguračních tříd e-*PFL*. Mezi hlavní metody patří *MakeConfigurationStaticDiagram*, *MakeConfigurationOverviewDiagram*, *MakeConfigurationOverviewDeployDiagram*, které zajišťují samotné vytvoření UML modelů a uložení do XMI souboru.

- **XMIDiagramConfiguration** - třída pracuje s konfiguračním souborem *e-PFL.diagram.config*, kde může uživatel částečně ovlivnit podobu výstupního XMI souboru.

V těle metody *Run* překladače e-PFL.exe byla přidána nová možnost konzolového přepínače (-g), který spustí generování konkrétních diagramů do souboru s názvem *[název pfl souboru].diagram.xmi*.

```
/*****************************************************************/
/********************** Generate XMI diagrams ************************/
if (generateUMLDiagrams)
{
   Console.WriteLine("Generate diagram file: " + fileName + ".diagram.xmi");

        XMIDiagramGenerator xmiGenerator = new XMIDiagramGenerator();
        xmiGenerator.MakeDiagrams();
        xmiGenerator.XMI.Save(fileName + ".diagram.xmi");
}

Console.WriteLine("Done.");
```

Výpis 7: Integrace XMILibrary do metody *Run* v e-*PFL*

Vygenerovaný UML model byl pro přehlednost rozdělen do balíčků (packages), aby bylo docíleno lepší přehlednosti a usnadnila se také následná manipulace importovaného modelu v CASE nástrojích. Hiearchická struktura obsahuje tyto balíčky:

- E-PFL
  - Configuration
    - Base diagram
    - Configuration diagram
    - Overview class diagram
    - Overview deployment diagram
  - DataTypes

Kořenem je element nazvaný *E-PFL*. Jedná se o element typu *„UML:Model“*, kterým začíná každy model uložený v XMI. Balík *Configuration* obsahuje všechny modely vztahující se ke konfiguraci a designu vestavného systému. Třídní diagram obsažený v sekci *Base diagram* je technicky potřebný pro vytvoření instancí tříd v objektovém diagramu nazvaném *Configuration diagram*. Ten zachycuje startovní konfiguraci vestavného systému. *Overview class diagram* nám dává celkovou představu o designu a jednotlivých prvcích navrhovaného systému, jako jsou jeho komponenty, jednotky, vstupy a výstupy. *Overview deployment diagram* je druhou variantou zobrazení předešlého případu. Jen jsou použity prvky z „deployment“ diagramu tj. component, device, což nám vůči předešlému eliminovalo počet prvků a celkově zjednodušilo pohled. V balíku *Data types* jsou definovány všechny datové typy, na které se můžou odkazovat jednotlivé UML „typové“ elementy navrhovaného modelu. XMILibrary obsahuje mimo jiné také podporu pro přidání vlastního datového typu. Tento datový typ je pak automaticky přidán do stejného balíku.

5.3.2 Integrace generování diagramů popisující pohled na dynamické aspekty systému

Diagramy, které jsou vhodné pro zobrazení těchto aspektů systému mohou být např. *Activity diagram*, *Statechart diagram*, *Sequence diagram*. Jak už bylo zmíněno, překladač e-PFL vygeneruje .cs soubor obsahující třídy a zdrojový kód, který popisuje vzhled a funkčnost navrhovaného vestavného systému. Pokud chceme zachytit dynamické aspekty simulovaného vestavěného systému, musíme upravit část překladače, který se stará o vygenerovaní výsledného kódu. Cílem je do procesu generace výstupního .cs souboru vložit také vlastní kód, zajišťující generování dat pro vytvoření UML modelu.

V zásadě máme tyto možnosti:

1. Při simulaci můžeme zaznamenat jednotlivé akce a průchody funkcemi do nějaké vhodné struktury, kupříkladu textový sturkturovaný soubor, nebo XML soubor. Tyto data by se dále načetly a zpracovaly v nějaké vlastní aplikaci, kde by se na základě předem specifikovaných kriterií transformovaly do konkrétních UML modelů a uložily v podobě XMI souboru. K tomuto účelu by šlo i s největší pravděpodobností využít nějaký hotový nástroj (knihovnu) pro tracing v .NET. Pak by se generovaný kód pouze rozšířil o tzv. „trace pointy“ a nad výstupním souborem by se vytvořil vlastní nástroj pro zpracování těchto dat.

2. Dynamicky připojit knihovnu *XMILibrary.dll* a vytvářet modely rovnou za běhu, voláním jednotlivých funkcí, které poskytuje tato knihovna.

První řešení je vemi komplexní a je i připraveno na různé pozdější analýzy. Záleží samozřejmě do jaké hloubky a jaká data by nám generovaný soubor dával k dispozici. Zde je důležitým prvkem vhodně navrhout samotnou strukturu takového souboru. Druhé řešení je přímočarejší a nezohledňuje tolik budoucí potřeby. Nakonec byl zvolen druhý způsob, který je pro naše edukativní účely dostatečný a z hlediska již existující knihovny *XMILibrary* i jednodušší.

Při další analýze jsem narazil na tyto problémy:

- Embedded systém, který je kompilovaný do spustitelného souboru z e-$\mathcal{PFL}$ je v podstatě distribuován volně a samostatně. Proto byla zvolena strategie, že pokud není ve stejné cestě umístěná i assembly *XMILibrary.dll*, tak funkce generování diagramů bude vypnuta. V případě, že assemby existuje, je dynamicky nahrána a jednotlivé funkce jsou volány pomocí .NET reflexi. K tomuto účelu slouží níže uvedená třída *XMILibraryWrapper*, která „zabaluje“ volání jednotlivých funkcí knihovny. Uvedená třída podporuje v současné době pouze diagram aktivit a je fyzicky umístěná v *resource* souboru překladače e-$\mathcal{PFL}$, který ji automaticky vkládá do generovaného .cs souboru spolu s ostatními třídami při překladu.

- Jednotlivé jednotky (devices) popsaného vestavného systému běží ve vlastních vláknech a mohou pracovat i konkurenčně (fair a unfair režim). Není proto snadné zvolit

strategii vytvoření takového diagramu, aby smysluplně zachytil nějakou činnost. Další problém je, volání jednotlivých funkcí „wrapperu“ z jednotlivých vláken. To bylo řešeno použitím návrhového vzoru typu *„thread safe Singleton“* a použitím zámků uvnitř volání jednotlivých funkcí.

- U většiny modelovaných vestavných systémů pracují jednotky v nekonečném cyklu a provádějí akce, dokud uživatel nezastaví program stiskem klávesy. Takto generovaný diagram by byl zbytečně rozsáhlý a neužitečný. Proto je třeba stanovit tzv. počet strojových cyklů, které daná jednotka provede a pak se ukončí.

Kromě vložení vlasní třídy *XMILibraryWrapper* do generovaného kódu, byly i modifikovány existující třídy *SimulatorGenerator* a *MicroNETGenerator* tak, aby docházelo k automatické konstrukci aktivitního diagramu. Tuto podporu může uživatel vyvolat novým konzolovým přepínačem *-g[počet strojových cyklů jednotek]*.

```
namespace XMILibrary
{
    using System.IO;
    using System.Reflection;

    public class XMILibraryWrapper
    {
        private static volatile XMILibraryWrapper instance;
        private static object syncRoot = new Object();
        private static Assembly assembly;
        private static object xmi;
        private static Type xmiType;
        private static object activity ;
        private string path;

        private XMILibraryWrapper()
        {
            path = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
            string assemblyFileName = Path.Combine(path, @"XMILibrary.dll");

            if (File.Exists(assemblyFileName))
            {
                assembly = Assembly.LoadFile(assemblyFileName);

                xmiType = assembly.GetType("XMILibrary.XMI");
                xmi = assembly.CreateInstance("XMILibrary.XMI", true, BindingFlags.Default, null,
                    new[] { "Dynamic model" }, null, null);
                activity = assembly.CreateInstance("XMILibrary.Activities.Activity", true,
                    BindingFlags.Default, null, new[] { "Activity diagram" }, null, null);

                PropertyInfo propModel = xmiType.GetProperty("Model");
                var model = propModel.GetValue(xmi, null);

                AddItemList(model, "OwnedMember", activity);
            }
            else
```

```
    {
        assembly = null;
    }
}

public static XMILibraryWrapper Instance
{
    get
    {
        if (instance == null)
        {
            lock (syncRoot)
            {
                instance = new XMILibraryWrapper();
            }
        }
        return instance;
    }
}

public bool IsActivityHandlerEnabled
{
    get
    {
        return assembly != null;
    }
}

public object AddInitialNode(string name)
{
    if (!IsActivityHandlerEnabled) return null;

    lock(this)
    {
        return CreateNode("XMILibrary.Activities.InitialNode", name);
    }
}

public object AddForkNode(string name, object comesFrom)
{
    if (!IsActivityHandlerEnabled) return null;

    lock(this)
    {
        var fork = CreateNode("XMILibrary.Activities.ForkNode", name);
        AddControlFlow(comesFrom, fork, string.Empty);
        return fork;
    }
}

public object AddJoinNode(string name)
{
    if (!IsActivityHandlerEnabled) return null;
```

```
        lock (this)
        {
            var join = CreateNode("XMILibrary.Activities.JoinNode", name);
            return join;
        }
    }

    public void AddControlFlow(object nodeFrom, object nodeTo, string guard)
    {
        MethodInfo methodInfo = nodeFrom.GetType().GetMethod("AddControlFlow", new Type
            [] { assembly.GetType("XMILibrary.Activities.ActivityNode"), typeof(string) });
        object o = methodInfo.Invoke(nodeFrom, new[] { nodeTo, guard });
        AddItemList(activity, "Edge", o);
    }

    public object AddAction(string name, object comesFrom, string guard)
    {
        if (!IsActivityHandlerEnabled) return null;

        lock (this)
        {
            var activityAction = CreateNode("XMILibrary.Activities.ActivityAction", name);
            AddControlFlow(comesFrom, activityAction, guard);
            return activityAction;
        }
    }

    public object AddActivityFinalNode(string name, object comesFrom)
    {
        if (!IsActivityHandlerEnabled) return null;

        lock (this)
        {
            var finalNode = CreateNode("XMILibrary.Activities.ActivityFinalNode", name);
            AddControlFlow(comesFrom, finalNode, string.Empty);
            return finalNode;
        }
    }

    public void AddItemList(object obj, string propName, object item)
    {
        lock (this)
        {
            PropertyInfo propInfo = obj.GetType().GetProperty(propName);
            object list = propInfo.GetValue(obj, null);
            MethodInfo m = list.GetType().GetMethod("Add");
            m.Invoke(list, new[] { item });
        }
    }

    public object CreateNode(string type, string name)
    {
        Type initNodeType = assembly.GetType(type);
        object initNodeInstance = Activator.CreateInstance(initNodeType, name);
```

```
            AddItemList( activity , "Node", initNodeInstance);
            return initNodeInstance;
        }

        public void SaveDiagram(string fileName)
        {
            fileName = Path.Combine(path, fileName);
            MethodInfo saveMethod = xmiType.GetMethod("Save");
            saveMethod.Invoke(xmi, new object[] { fileName });
        }
    }
}
```

Výpis 8: XMILibrary wrapper

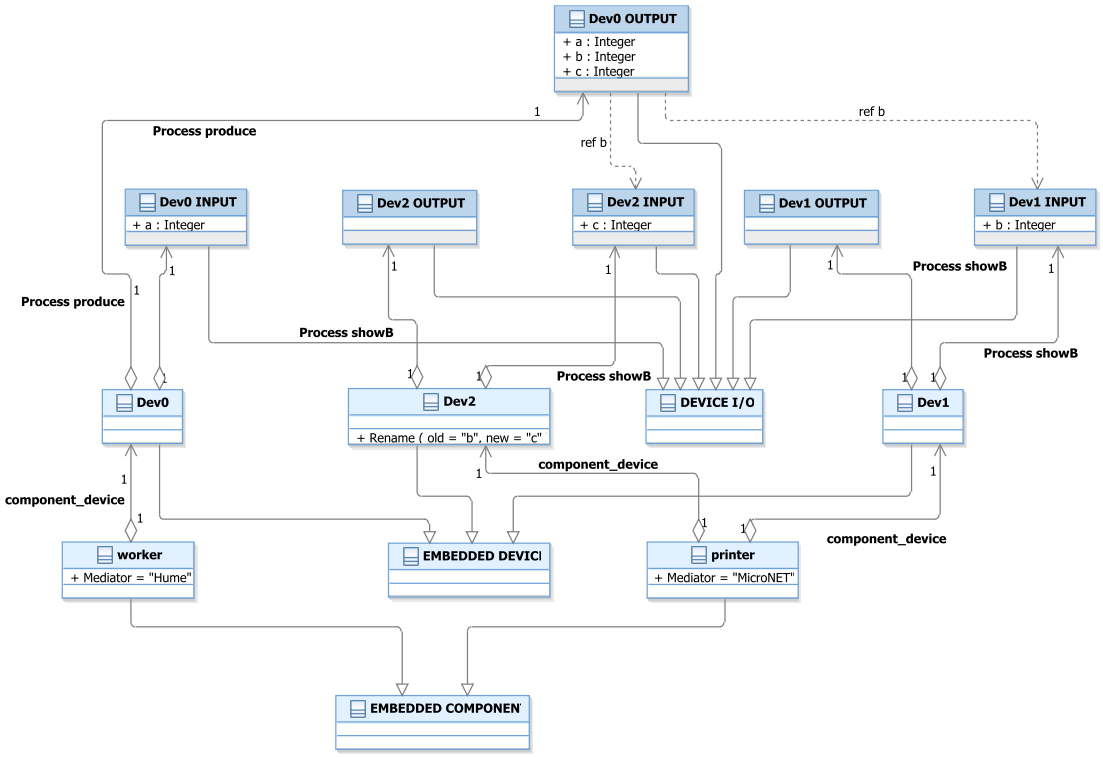
5.4 Demonstrace možností generovaných diagramů z e-$\mathcal{PFL}$

Zde je vybraná ukázka automaticky generovaných diagramů z jednoduché řešené e-$\mathcal{PFL}$ úlohy za pomocí integrované knihovny *XMILibrary.dll*.

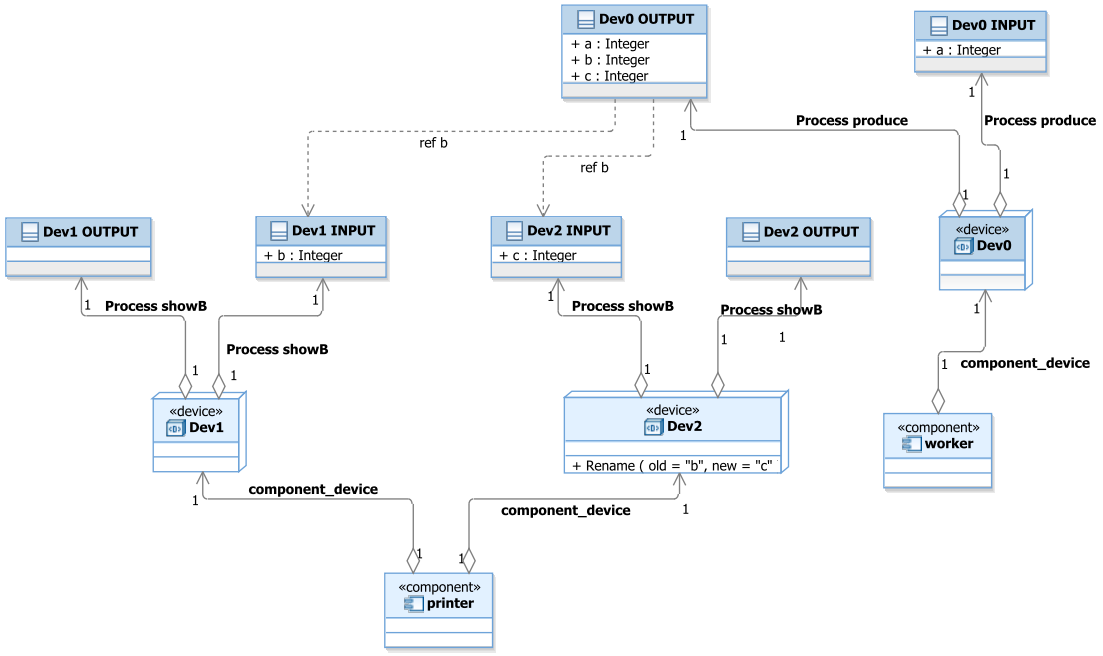
Poznámka: Více příkladů úloh e-$\mathcal{PFL}$ a generovaných komplexních diagramů je přiloženo na CD.

```
import "prelude.pfl"

produce :: a Integer -> (a Integer, b Integer, c Integer)
produce x = (x+1, x, x)

work :: Device
work = Process produce

showB :: b Integer -> ()
showB x = writeLine (show x)

printer :: Device
printer = Process showB

cmp1 :: EmbSystem
cmp1 = EmbComponent "worker" Hume

cmp2 :: EmbSystem
cmp2 = EmbComponent "printer" MicroNET

annotation :: Annotation
annotation = rename "b" "c"

main = (startDevice work cmp1 [])`bl`(startDevice printer cmp2 [])`bl`(startDevice printer cmp2 [
    annotation])
```

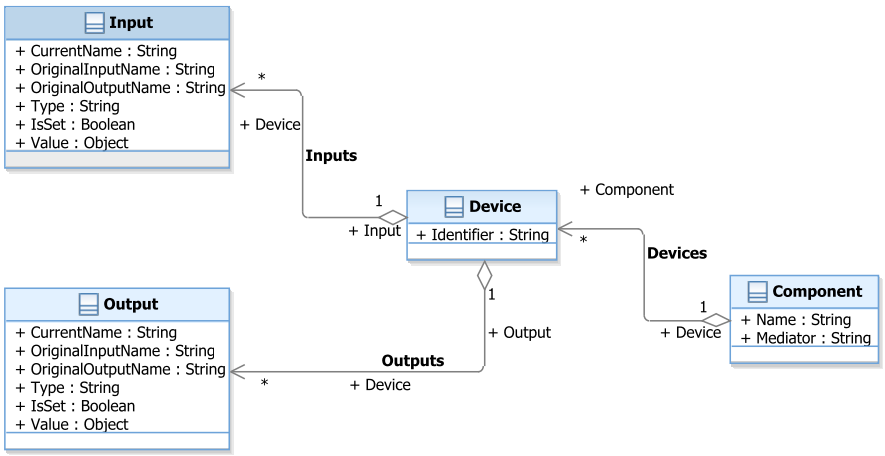
Výpis 9: Příklad úlohy v e-$\mathcal{PFL}$



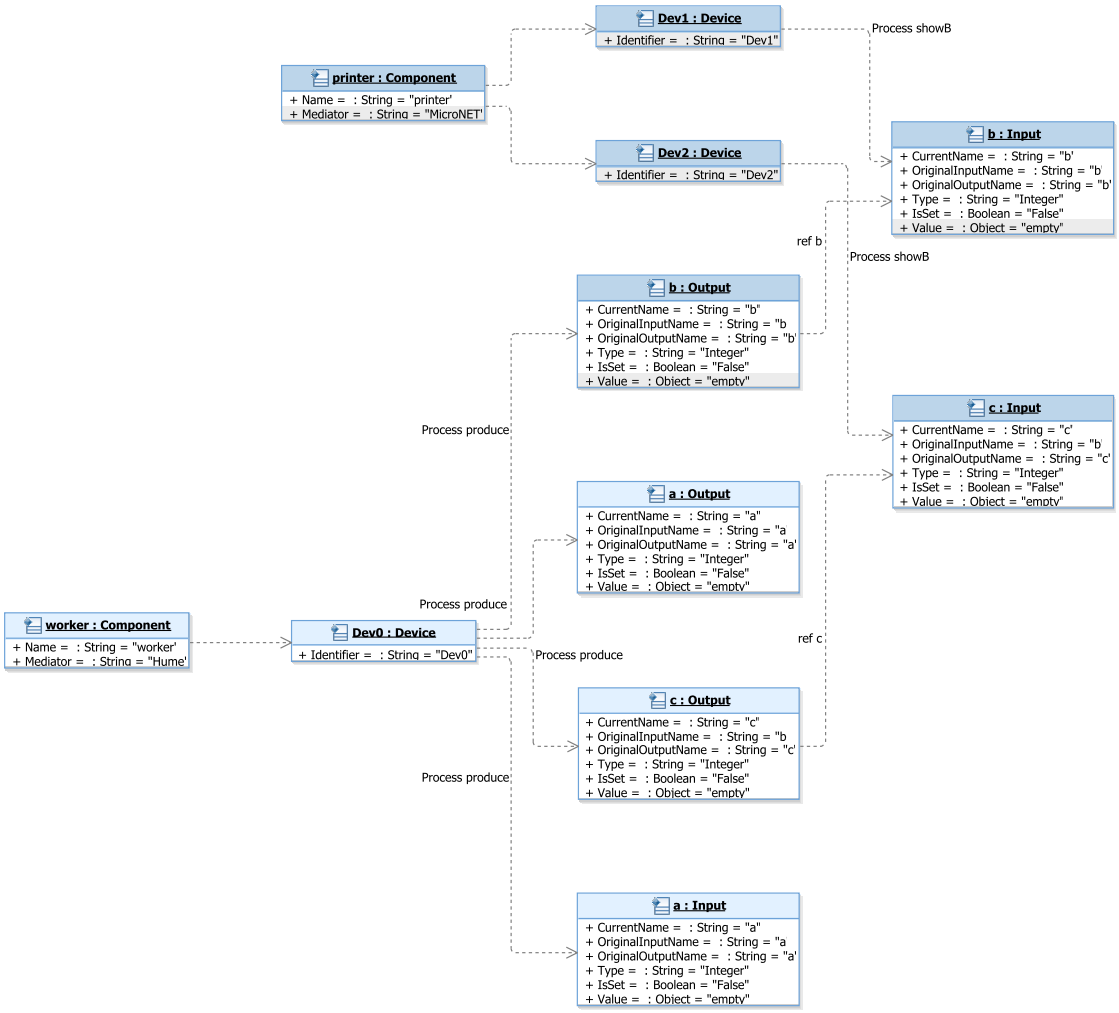
Obrázek 10: Overview diagram



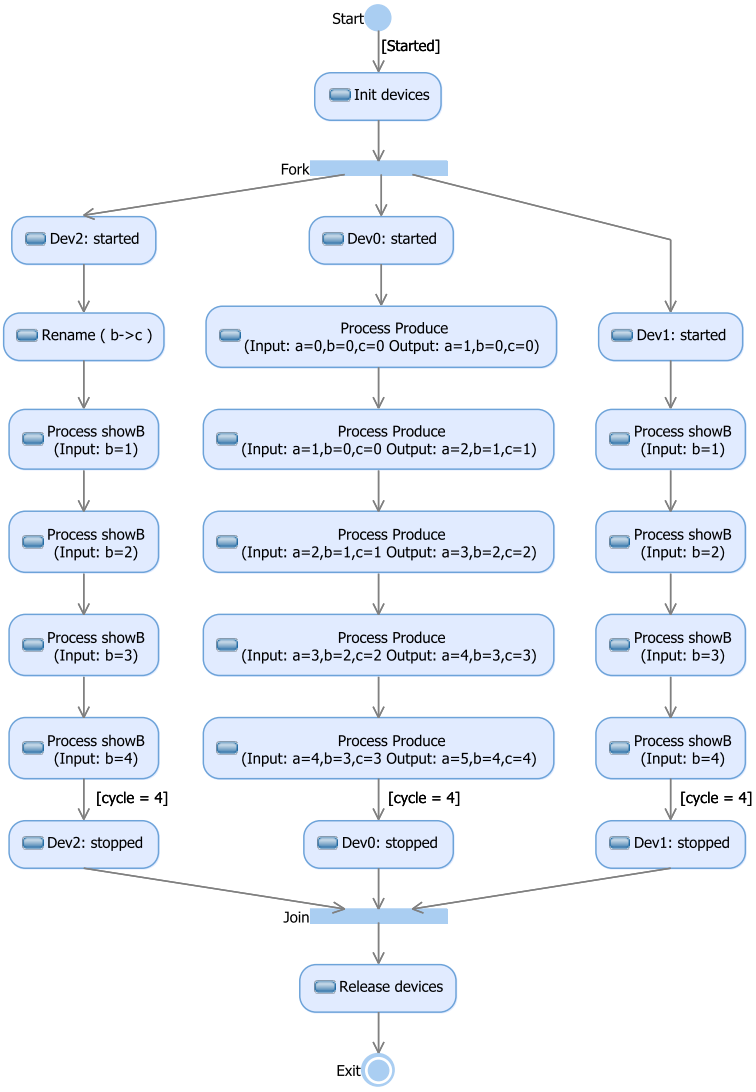
Obrázek 11: Overview diagram (deployment)



Obrázek 12: Configuration base diagram



Obrázek 13: Configuration diagram



Obrázek 14: Activity diagram

6 Závěr

V předložené diplomové práci byly prozkoumány možnosti využití UML pro popis vestavěných systémů a také vestavný procesně funkcionální jazyk e-*PFL*. Dále byl analyzován a realizován nástroj (konkrétně .NET assembly), umožňující generovat UML modely do *XML Metadata Interchange* specifického formátu souboru, který může být následně načten v CASE nástrojích a zpracován do podoby konkrétních UML diagramů. Vlastní nástroj byl vhodně integrován do stávajícího překladače e-*PFL* tak, aby bylo možné automaticky generovat diagramy popsaného vestavěného systému a zvýšila se tak jeho srozumitelnost. Generované diagramy zahrnují statický pohled na celkovou strukturu systému v podobě třídního diagramu a diagramu nasazení, dále objektový diagram zachycující startovní konfiguraci a diagram aktivit pro reprezentaci procesů a manipulaci se vstupy a výstupy při relném běhu. Součástí práce jsou také příklady demonstrující možnosti realizovaného nástroje.

Popis komplexních vestavěných systémů s pomocí UML není zcela snadná disciplína, speciálně pak real-time systémy, kde je potřeba v návrhu zachytit zprávy, signály, synchronizaci, časové podmínky, komunikační kanály, konkurenční podmínky, HW i SW prvky atd. Ačkoli je UML primárně určen pro modelování objektově orientovaného (nebo tomuto paradigmatu blízkého) systému, poskytuje sám o sobě dostatečný komfort pro popis vestavěných systémů. V případě složitějších analýz systému jsou k dipozici některé z UML profilů RT-UML, MARTE, SysML atd., které rozšiřují modelovací jazyk o specifickou sémantiku a samotná specifikace často sebou také přináší popis jistých analytických postupů a vhodnost použití jednotlivých diagramů. Přesto může analytik narazit na jistá omezení a nedostatky. Pak lze UML kombinovat i s jinými nástroji jako je např. *Hardware Description Language* apod. Částečně je i na týmové dohodě, jak budou dané zápisy strukturovány a interpretovány.

Po průzkumu technologií pro přenos UML modelů můžu tvrdit, že XMI (XML Metadata Interchange) je v současné době jediný životaschopný zavedený standard, který je schopen splnit své poslání, jako prostředek výměny objektů UML modelu a metadat informací mezi CASE nástroji. Přes některé problémy s kompatibilitou v CASE nástrojích, které se ovšem konsorcium OMG s výrobci CASE nástrojů snaží odstranit, lze říct, že tento standard budu podporován i v budoucnu.

Procesně funkcionalní jazyk e-*PFL* představuje zajimavou možnost a vlastnosti v oblasti modelování vestavěných systémů pomocí hybridního funkcionálního jazyka, vycházející z čistě funkcionálního jazyka Haskell. e-*PFL* je v současné době stále ve fázi vývoje, ale přesto lze namodelovat poměrně komplexní systém. Jedním z příkladu může být zdrojový soubor vending.pfl, který je přiložen na CD.

6.1 Přínos diplomové práce

Tato práce nepřináší žádné nové vědecké poznatky. Konkrétním výstupem a hlavním přínosem diplomové práce, bylo vytvořit .NET knihovnu pro generování UML diagramů a integrovat ji s hybridním procesně funkcionálním jazykem e-*PFL*, určeným pro modelo-

vání vestavných systémů v prvních fázích vývoje. Úkolem těchto diagramů je zpřehlednit a usnadnit pochopení navrhovaného systému.

Hlavním osobním přínosem bylo obecně se seznámit s vestavěnými systémy, možnostmi jejich modelování jak pomocí UML, tak i pomocí procesně funkcionálního jazyka e-*PFL*. Vedle nabytí teoretických znalostí jsem si i prakticky osvojil práci s celou škálou SW nástrojů. Při vlastní relizaci jsem pak zúročil dosavadní profesionální praxi v SW firmě a propojil ji s teoretickými vědomostmi získanými při dálkovém studiu tohoto univerzitního oboru.

6.2 Možnosti dalšího rozšíření práce

Na základě aktuálního stavu diplomové práce lze navrhnout další možnosti rozšíření výzkumu, jak po stránce teoretické, tak i praktické:

- Assembly *XMILibrary* by mohla být rozšířena o další elementy, jak definuje specifikace UML např. *Stavový diagram*, *Sekvenční diagram*, *Diagram komunikace*, *Diagram časování* atd.
- Stále existuje určitý volný prostor pro vytipování a analýzu možností generování dalších diagramů z e-*PFL*, zejména pak v oblasti popisující dynamické aspekty systému.
- U složitějších generovaných diagramů se ztrácí markantně přehlednost, proto by bylo žádoucí parametrizovat v konfiguračním souboru *e-PFL.diagram.config* úroveň jejich složitosti a vygenerovat méně detailní pohledy, nebo diagramy parciálně rozdělit podle nějakého klíče apod.
- Hlouběji se ponořit do modelování vestavěných systémů pomocí UML a zaměřit se přitom na profil MARTE, který je v současné době nejkomplexnější pro tyto účely.

Tomáš Huplík

7 Literatura

[1] Arlow, J.; Neustadt, I.: *UML 2 and the Unified Process Second Edition*. USA: Addison Wesley, 2005, ISBN 0-321-32127-8.

[2] Automa: Vestavěné systémy. [online], [cit. 2011-21-11].
URL `http://www.odbornecasopisy.cz/index.php?id_document=28629`

[3] Beneš, M.: Funkcionální programování. [online], [cit. 2011-21-11].
URL `http://www.cs.vsb.cz/behalek/education/pp/texty/fp/index.html`

[4] Běhálek, M.: *Vestavný procesně funkcionalní jazyk*. Disertační práce, VŠB-TU Ostrava, 2009.

[5] Colin, W.: *Embedded software: The works*. Burlington, USA: Elsevier Inc., 2006, ISBN 0-7506-7954-9.

[6] Douglass, B., Powel: *Real-Time UML workshop for embedded systems*. Burlington, USA: Elsevier Inc., 2006, ISBN 0-7506-7906-9.

[7] Douglass, B.-P.: *Real Time UML: Advances in The UML for Real-Time Systems, Third Edition*. USA: Addison Wesley, 2004, ISBN 0-321-16076-2.

[8] Duc, B., Minh: *Real-Time object uniform methodology with UML*. New York, NY, USA: Springer Publishing, 2007, ISBN 978–1–4020–5976–6.

[9] Eriksson, H.-E.; Penker, M.; Lyons, B.; aj.: *UML 2 Toolkit*. Indianapolis, Indiana: Wiley Publishing Inc., 2004, ISBN 0-471-46361-2.

[10] Fowler, M.; Wesley, A.: *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. USA: Addison Wesley, 2003, ISBN 0-321-19368-7.

[11] Ganssle, J.; Barr, M.: *Embedded Systems Dictionary*. USA: CMP Books, 2003, ISBN 1578201209.

[12] Gomes, L.; Fernandes, J., M.: *Behavioral Modeling for Embedded Systems and Technologies - Applications for Design and Implementation.* Redmond, USA: IGI Global, 2010, ISBN 978-1-60566-751-5.

[13] Grant, M.: UML for Embedded Systems Specification and Design: Motivation and Overview. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, Washington, DC, USA: IEEE Computer Society, 2002, ISBN 0-7695-1471-5, str. 3.

[14] Grose, T., J.; Doney, G., C.; Brodsky, S., A.: *Mastering XMI - Java Programming with XMI, XML and UML*. USA: John Wiley & Sons, 2002, ISBN 0471384291.

[15] Miles, R.; Hamilton, K.: *Learning UML 2.0.* O'Reilly Media, 2006, ISBN 0596009828.

[16] OMG: Meta Object Facility (MOF) Core Specification ver. 2.0. [online], [cit. 2011-21-11].
URL http://www.omg.org/spec/MOF/2.0/PDF

[17] OMG: MOF 2.0/XMI Mapping specification ver. 2.1.1. [online], [cit. 2011-21-11].
URL http://www.omg.org/spec/XMI/2.1/PDF

[18] OMG: Object Constraint Language ver. 2.2. [online], [cit. 2011-21-11].
URL http://www.omg.org/spec/OCL/2.2/PDF

[19] OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems ver. 1.1. [online], [cit. 2011-21-11].
URL http://www.omg.org/spec/MARTE/1.1/PDF

[20] OMG: Unified Modeling Language Infrastructure ver. 2.1.2. [online], [cit. 2011-21-11].
URL http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF

[21] OMG: Unified Modeling Language Superstructure ver. 2.1.2. [online], [cit. 2011-21-11].
URL http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF

[22] Thompson, D.: *Embedded Programming with the Microsoft .NET Micro Framework (Pro-Developer)*. Redmond, WA, USA: Microsoft Press, 2007, ISBN 0735623651.

[23] Wikipedia: CASE nástroje. [online], [cit. 2011-21-11].
URL http://cs.wikipedia.org/wiki/CASE_n%C3%A1stroje

[24] Wikipedia: Embedded system. [online], [cit. 2011-21-11].
URL http://en.wikipedia.org/wiki/Embedded_systems

[25] Wikipedia: Hardware description language. [online], [cit. 2011-21-11].
URL http://en.wikipedia.org/wiki/Hardware_description_language

A Obsah přiloženého CD

Na CD přiloženém k této práci se nacházejí následující složky:

- **Examples** - obsahuje souhrn příkladů v e-*PFL*, včetně vygenerovaných XMI souborů a UML diagramů v pdf formátu.
- **Sources**
  - **E-PFL** - obsahuje zdrojový kód nástroje e-*PFL*, rozšířené o integraci vlastní assembly *XMILibrary* pro generování diagramů.
  - **XMILibrary** - obsahuje zdrojový kód vlastní vytvořené assembly *XMILibrary* pro práci s UML diagramy a testovací aplikaci *TestXMI*.
- **Text** - obsahuje samotný text diplomové práce vytvořený v LaTeX.

B Výpisy XMI k příkladům

```
<?xml version="1.0" encoding="utf-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://
    schema.omg.org/spec/UML/2.1.1">
  <xmi:Documentation contact="tomas.huplik@gmail.com" exporter="XMI Factory" exporterVersion
      ="1" />
  <uml:Model xmi:id="ea05e00d-2416-47d9-aa72-7a4723fad34b" name="TestModel" xmi:type=
      "uml:Model" visibility="public">
    <packagedElement xmi:id="02546fd9-0ea0-4ceb-89f9-4da5d7ceea55" name="Class3"
        xmi:type="uml:Class" visibility="public" isAbstract="false">
      <ownedAttribute xmi:id="8f50a410-7c95-48cb-bcd3-38c27c9f14c3" name="Class2"
          xmi:type="uml:Property" type="fa9c785c-cc68-4e52-8674-61393dba103d" visibility="
          public" isDerived="false" isID="false" isReadonly="false" isUnique="true" aggregation="
          composite">
        <association xmi:idref="d092ddf4-009e-4a16-8543-0aa6d4a4bdad" />
        <lowerValue xmi:id="dd970ad9-33ff-4723-a1a4-389ea1a4c2ce" visibility="public"
            xmi:type="uml:LiteralInteger" value="1" />
        <upperValue xmi:id="da4b2448-f643-4c20-9bc4-bea4b3354505" visibility="public"
            xmi:type="uml:LiteralUnlimitedNatural" value="2" />
      </ownedAttribute>
    </packagedElement>
    <packagedElement xmi:id="d092ddf4-009e-4a16-8543-0aa6d4a4bdad" name="association
        " xmi:type="uml:Association" visibility="public">
      <memberEnd xmi:idref="8f50a410-7c95-48cb-bcd3-38c27c9f14c3" />
      <memberEnd xmi:idref="9439b0ae-b4b8-46e1-b044-4ec081b3ff9a" />
      <ownedEnd xmi:id="9439b0ae-b4b8-46e1-b044-4ec081b3ff9a" name="Class3" xmi:type=
          "uml:Property" type="02546fd9-0ea0-4ceb-89f9-4da5d7ceea55" visibility="public"
          isDerived="false" isID="false" isReadonly="false" isUnique="true">
        <association xmi:idref="d092ddf4-009e-4a16-8543-0aa6d4a4bdad" />
      </ownedEnd>
    </packagedElement>
    <packagedElement xmi:id="fa9c785c-cc68-4e52-8674-61393dba103d" name="Class2"
        xmi:type="uml:Class" visibility="public" isAbstract="false">
      <ownedAttribute xmi:id="7a2fd889-2e85-42e3-949a-196fcc29749c" name="Property2"
          xmi:type="uml:Property" type="64553c62-bdb7-4a07-81cf-4f9ee7e9f922" visibility="
          public" isDerived="false" isID="false" isReadonly="false" isUnique="true" />
      <generalization xmi:id="0ce8dc36-553f-4c59-be25-fcf096312480" xmi:type="
          uml:Generalization" isSubstitutable="true" general="4b3a7f79-2a7f-43f9
          -9878-792159b852ae" />
    </packagedElement>
    <packagedElement xmi:id="4b3a7f79-2a7f-43f9-9878-792159b852ae" name="Class1"
        xmi:type="uml:Class" visibility="public" isAbstract="true">
      <clientDependency xmi:idref="1aa5f46d-ede6-4d18-b3e8-8dba2d7c5c23" />
      <ownedComment xmi:id="3fa58787-6658-4c1a-a375-558c8e4b716b" xmi:type="
          uml:Comment" body="Abstract class">
        <annotatedElement xmi:idref="4b3a7f79-2a7f-43f9-9878-792159b852ae" />
      </ownedComment>
      <ownedAttribute xmi:id="bbba6793-78e9-4dd4-873d-e3cf80fc1fbd" name="Property1"
          xmi:type="uml:Property" type="c2beabb0-5e41-45f3-969e-386d8a0eda03" visibility="
          public" isDerived="false" isID="false" isReadonly="false" isUnique="true" />
      <ownedOperation xmi:id="22e37085-580f-4f59-a15a-718b84af9fde" name="Operation1"
          xmi:type="uml:Operation" visibility="public">
```

```
            <ownedParameter xmi:id="e0884064-3edc-476e-b986-7a47dfb0b61d" name="param1"
                 xmi:type="uml:Parameter" type="c947c57f-4709-4a32-b3c5-8b0ae8258a73"
                direction="in">
              <upperValue xmi:id="8705829b-27bd-4e90-968f-170696ac6af6" visibility="public"
                   xmi:type="uml:LiteralUnlimitedNatural" value="1" />
            </ownedParameter>
            <ownedParameter xmi:id="ee9263ef-d12f-4f61-be4a-18956f8cd0a3" name="param2"
                 xmi:type="uml:Parameter" type="c947c57f-4709-4a32-b3c5-8b0ae8258a73"
                direction="return">
              <upperValue xmi:id="7867c7af-3c51-41bd-87c6-111c0e983277" visibility="public"
                   xmi:type="uml:LiteralUnlimitedNatural" value="1" />
            </ownedParameter>
          </ownedOperation>
          <interfaceRealization xmi:id="1aa5f46d-ede6-4d18-b3e8-8dba2d7c5c23" xmi:type="
              uml:InterfaceRealization" contract="f166f8c5-5c72-425e-a45a-e10d34015d9d">
            <supplier xmi:idref="f166f8c5-5c72-425e-a45a-e10d34015d9d" />
            <client  xmi:idref="4b3a7f79-2a7f-43f9-9878-792159b852ae" />
          </interfaceRealization>
        </packagedElement>
        <packagedElement xmi:id="f166f8c5-5c72-425e-a45a-e10d34015d9d" name="Inteface1"
             xmi:type="uml:Interface" visibility="public" isAbstract="false">
          <ownedAttribute xmi:id="842fd9fb-8034-4c61-afac-3df8f0715631" name="Property1"
               xmi:type="uml:Property" type="c2beabb0-5e41-45f3-969e-386d8a0eda03" visibility="
               public" isDerived="false" isID="false" isReadonly="false" isUnique="true" />
        </packagedElement>
        <packagedElement xmi:id="ba94e851-a1de-4867-958b-ce6b41ce2665" name="Datatypes"
             xmi:type="uml:Package" visibility="public">
          <packagedElement xmi:id="c2beabb0-5e41-45f3-969e-386d8a0eda03" name="Integer"
               xmi:type="uml:DataType" visibility="public" />
          <packagedElement xmi:id="64553c62-bdb7-4a07-81cf-4f9ee7e9f922" name="Boolean"
               xmi:type="uml:DataType" visibility="public" />
          <packagedElement xmi:id="c947c57f-4709-4a32-b3c5-8b0ae8258a73" name="String"
               xmi:type="uml:DataType" visibility="public" />
          <packagedElement xmi:id="c4f4ba56-0d0a-4ea4-bbdf-ed55197a2721" name="
               UnlimitedNatural" xmi:type="uml:DataType" visibility="public" />
          <packagedElement xmi:id="1d9f39a3-99db-499c-a53c-36beceb3dca6" name="Object"
               xmi:type="uml:DataType" visibility="public" />
        </packagedElement>
      </uml:Model>
    </xmi:XMI>
```

Výpis 10: Výstupní soubor XMI generovaný z výpisu č.2

```
    <?xml version="1.0" encoding="utf-8"?>
    <xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://
        schema.omg.org/spec/UML/2.1.1">
      <xmi:Documentation contact="tomas.huplik@gmail.com" exporter="XMI Factory" exporterVersion
          ="1" />
      <uml:Model xmi:id="526b9308-f22c-4bb7-b418-fe13c67c2d17" name="TestModel" xmi:type="
          uml:Model" visibility="public">
        <packagedElement xmi:id="8a511f21-eb54-4d9a-b3c9-785a54cb4dcb" name="Activity
            diagram" xmi:type="uml:Activity" visibility="public">
          <edge xmi:id="78771c06-aa62-48ea-b8fc-29b1d73705df" xmi:type="uml:ControlFlow">
```

```
  <source xmi:idref="4c7c1a9e-fded-4478-8397-f87564d3b052" />
  <target xmi:idref="677dd867-fae5-4d7c-9120-95db32d7e0ad" />
</edge>
<edge xmi:id="b090bf12-77f5-4e1b-94d1-f26f8d70c764" xmi:type="uml:ControlFlow">
  <source xmi:idref="677dd867-fae5-4d7c-9120-95db32d7e0ad" />
  <target xmi:idref="bee649f2-f8f6-4ae6-a12d-7f2e659aa1ac" />
</edge>
<edge xmi:id="23ee20fc-bf36-4eee-9a79-344113d4e544" xmi:type="uml:ControlFlow">
  <source xmi:idref="bee649f2-f8f6-4ae6-a12d-7f2e659aa1ac" />
  <target xmi:idref="fa375bb0-fd06-4088-b854-afc8da081487" />
  <guard xmi:id="6dcb23e8-3809-4b63-86e3-74517984c211" visibility="public" xmi:type=
      "uml:LiteralString" value="false" />
</edge>
<edge xmi:id="ec26ce99-1dd5-4199-8680-2a77adfeff14" xmi:type="uml:ControlFlow">
  <source xmi:idref="fa375bb0-fd06-4088-b854-afc8da081487" />
  <target xmi:idref="8af94323-6c3c-4aeb-99ec-b88537202909" />
</edge>
<edge xmi:id="43303ea9-125b-4a33-86cb-c6b330100e12" xmi:type="uml:ControlFlow">
  <source xmi:idref="8af94323-6c3c-4aeb-99ec-b88537202909" />
  <target xmi:idref="c76d6a59-bee1-4d1f-adcf-c6bcbe3049fa" />
</edge>
<edge xmi:id="1053b629-db94-461a-b0fe-a934c471cca7" xmi:type="uml:ControlFlow">
  <source xmi:idref="8af94323-6c3c-4aeb-99ec-b88537202909" />
  <target xmi:idref="dcdf5c73-673a-4344-9c06-30baad5abde3" />
</edge>
<edge xmi:id="5b69d3fc-9756-429e-9680-b585b5063b91" xmi:type="uml:ControlFlow">
  <source xmi:idref="c76d6a59-bee1-4d1f-adcf-c6bcbe3049fa" />
  <target xmi:idref="f0e03086-af9c-4e2c-9635-80c9febacb48" />
</edge>
<edge xmi:id="583d3e30-2cc8-4503-b9f0-96647ca52cef" xmi:type="uml:ControlFlow">
  <source xmi:idref="dcdf5c73-673a-4344-9c06-30baad5abde3" />
  <target xmi:idref="f0e03086-af9c-4e2c-9635-80c9febacb48" />
</edge>
<edge xmi:id="73f58a24-f422-418d-8f19-9979b40564ec" xmi:type="uml:ControlFlow">
  <source xmi:idref="f0e03086-af9c-4e2c-9635-80c9febacb48" />
  <target xmi:idref="91c1661a-1fab-4691-9ea1-9a5db3a88bdf" />
</edge>
<edge xmi:id="55eb5b10-4045-4d88-bf49-5133e07230b2" xmi:type="uml:ControlFlow">
  <source xmi:idref="bee649f2-f8f6-4ae6-a12d-7f2e659aa1ac" />
  <target xmi:idref="91c1661a-1fab-4691-9ea1-9a5db3a88bdf" />
  <guard xmi:id="ef2905fb-8061-4615-8f59-cea25828a3b9" visibility="public" xmi:type="
      uml:LiteralString" value="true" />
</edge>
<edge xmi:id="ebaab1ff-0bd6-4c43-b0b6-de5cfd1bcfc0" xmi:type="uml:ControlFlow">
  <source xmi:idref="91c1661a-1fab-4691-9ea1-9a5db3a88bdf" />
  <target xmi:idref="da5b7333-a727-4860-b58b-1c56cc9de58c" />
</edge>
<node xmi:id="4c7c1a9e-fded-4478-8397-f87564d3b052" name="Start" xmi:type="
    uml:InitialNode" visibility="public">
  <outgoing xmi:idref="78771c06-aa62-48ea-b8fc-29b1d73705df" />
</node>
<node xmi:id="677dd867-fae5-4d7c-9120-95db32d7e0ad" name="Action1" xmi:type="
    uml:OpaqueAction" visibility="public">
  <incoming xmi:idref="78771c06-aa62-48ea-b8fc-29b1d73705df" />
```

```
      <outgoing xmi:idref="b090bf12-77f5-4e1b-94d1-f26f8d70c764" />
    </node>
    <node xmi:id="bee649f2-f8f6-4ae6-a12d-7f2e659aa1ac" name="x &gt; 10" xmi:type="
        uml:DecisionNode" visibility="public">
      <incoming xmi:idref="b090bf12-77f5-4e1b-94d1-f26f8d70c764" />
      <outgoing xmi:idref="23ee20fc-bf36-4eee-9a79-344113d4e544" />
      <outgoing xmi:idref="55eb5b10-4045-4d88-bf49-5133e07230b2" />
    </node>
    <node xmi:id="fa375bb0-fd06-4088-b854-afc8da081487" name="Action2" xmi:type="
        uml:OpaqueAction" visibility="public">
      <incoming xmi:idref="23ee20fc-bf36-4eee-9a79-344113d4e544" />
      <outgoing xmi:idref="ec26ce99-1dd5-4199-8680-2a77adfeff14" />
    </node>
    <node xmi:id="8af94323-6c3c-4aeb-99ec-b88537202909" name="Fork" xmi:type="
        uml:ForkNode" visibility="public">
      <incoming xmi:idref="ec26ce99-1dd5-4199-8680-2a77adfeff14" />
      <outgoing xmi:idref="43303ea9-125b-4a33-86cb-c6b330100e12" />
      <outgoing xmi:idref="1053b629-db94-461a-b0fe-a934c471cca7" />
    </node>
    <node xmi:id="c76d6a59-bee1-4d1f-adcf-c6bcbe3049fa" name="Action3" xmi:type="
        uml:OpaqueAction" visibility="public">
      <incoming xmi:idref="43303ea9-125b-4a33-86cb-c6b330100e12" />
      <outgoing xmi:idref="5b69d3fc-9756-429e-9680-b585b5063b91" />
    </node>
    <node xmi:id="dcdf5c73-673a-4344-9c06-30baad5abde3" name="Action4" xmi:type="
        uml:OpaqueAction" visibility="public">
      <incoming xmi:idref="1053b629-db94-461a-b0fe-a934c471cca7" />
      <outgoing xmi:idref="583d3e30-2cc8-4503-b9f0-96647ca52cef" />
    </node>
    <node xmi:id="f0e03086-af9c-4e2c-9635-80c9febacb48" name="Join" xmi:type="
        uml:JoinNode" visibility="public">
      <incoming xmi:idref="5b69d3fc-9756-429e-9680-b585b5063b91" />
      <incoming xmi:idref="583d3e30-2cc8-4503-b9f0-96647ca52cef" />
      <outgoing xmi:idref="73f58a24-f422-418d-8f19-9979b40564ec" />
    </node>
    <node xmi:id="91c1661a-1fab-4691-9ea1-9a5db3a88bdf" name="Action5" xmi:type="
        uml:OpaqueAction" visibility="public">
      <incoming xmi:idref="73f58a24-f422-418d-8f19-9979b40564ec" />
      <incoming xmi:idref="55eb5b10-4045-4d88-bf49-5133e07230b2" />
      <outgoing xmi:idref="ebaab1ff-0bd6-4c43-b0b6-de5cfd1bcfc0" />
    </node>
    <node xmi:id="da5b7333-a727-4860-b58b-1c56cc9de58c" name="Exit" xmi:type="
        uml:ActivityFinalNode" visibility="public">
      <incoming xmi:idref="ebaab1ff-0bd6-4c43-b0b6-de5cfd1bcfc0" />
    </node>
  </packagedElement>
  <packagedElement xmi:id="b4ae12a7-06d9-420d-8d7a-06393620df20" name="Datatypes"
      xmi:type="uml:Package" visibility="public">
    <packagedElement xmi:id="a9e27f95-8f9b-4ea4-9a73-83f13dc2507b" name="Integer"
        xmi:type="uml:DataType" visibility="public" />
    <packagedElement xmi:id="2c5c8006-1824-464d-bdad-e4b4ba59b9ea" name="Boolean"
        xmi:type="uml:DataType" visibility="public" />
    <packagedElement xmi:id="fd6e6e44-c52f-415b-abc9-b6d6ec2fbec5" name="String"
        xmi:type="uml:DataType" visibility="public" />
```

```
    <packagedElement xmi:id="3e6cfc52-d6e4-418c-93c3-6cd700f4159d" name="
        UnlimitedNatural" xmi:type="uml:DataType" visibility="public" />
    <packagedElement xmi:id="7f6aa60a-f436-4561-ae75-4e46a73e158e" name="Object"
        xmi:type="uml:DataType" visibility="public" />
  </packagedElement>
 </uml:Model>
</xmi:XMI>
```

Výpis 11: Výstupní soubor XMI generovaný z výpisu č.3

```
<?xml version="1.0" encoding="utf-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://
    schema.omg.org/spec/UML/2.1.1">
 <xmi:Documentation contact="tomas.huplik@gmail.com" exporter="XMI Factory" exporterVersion
     ="1" />
 <uml:Model xmi:id="610ab750-db4f-4d1e-880b-a9f2307dac9e" name="TestModel" xmi:type="
     uml:Model" visibility="public">
  <packagedElement xmi:id="8da953f0-ff41-4284-9193-adb21e4583c2" name="Artifact1"
      xmi:type="uml:Artifact" visibility="public" />
  <packagedElement xmi:id="cda625a5-78a6-4e8a-9969-c9f08152eec4" name="
      Component1" xmi:type="uml:Component" visibility="public" isAbstract="false"
      isIndirectlyInstantiated="true">
   <clientDependency xmi:idref="a4b3829f-c775-49aa-9311-fe0f95ef4146" />
   <ownedAttribute xmi:id="07a2b0ff-4243-4a41-914d-69ff5ad51f55" name="Property1"
       xmi:type="uml:Property" visibility="public" isDerived="false" isID="false" isReadonly="
       false" isUnique="true" />
   <ownedAttribute xmi:id="51f79210-061e-4d87-bf61-e9174eb1d2da" name="
       ExecutionEnvironment1" xmi:type="uml:Property" type="c74efc66-6972-41da-98e2-
       e5077ff0d0fd" visibility="public" isDerived="false" isID="false" isReadonly="false"
       isUnique="true">
    <association xmi:idref="f7923f74-91c4-4be5-bbbb-32b1cf4cefb8" />
   </ownedAttribute>
   <ownedOperation xmi:id="579e40e8-3db4-4bf7-8167-71dd6c38c0d1" name="Operation1
       " xmi:type="uml:Operation" visibility="public" />
  </packagedElement>
  <packagedElement xmi:id="a4b3829f-c775-49aa-9311-fe0f95ef4146" xmi:type="
      uml:Dependency">
   <supplier xmi:idref="2d2ef8bb-6029-4222-b546-cf132aa6808e" />
   <client  xmi:idref="cda625a5-78a6-4e8a-9969-c9f08152eec4" />
  </packagedElement>
  <packagedElement xmi:id="f7923f74-91c4-4be5-bbbb-32b1cf4cefb8" name="" xmi:type="
      uml:Association" visibility="public">
   <memberEnd xmi:idref="51f79210-061e-4d87-bf61-e9174eb1d2da" />
   <memberEnd xmi:idref="fbded481-465c-44c6-9737-3db8a3dfdb8a" />
   <ownedEnd xmi:id="fbded481-465c-44c6-9737-3db8a3dfdb8a" name="Component1"
       xmi:type="uml:Property" type="cda625a5-78a6-4e8a-9969-c9f08152eec4" visibility="
       public" isDerived="false" isID="false" isReadonly="false" isUnique="true">
    <association xmi:idref="f7923f74-91c4-4be5-bbbb-32b1cf4cefb8" />
   </ownedEnd>
  </packagedElement>
  <packagedElement xmi:id="2d2ef8bb-6029-4222-b546-cf132aa6808e" name="Device1"
      xmi:type="uml:Device" visibility="public" isAbstract="false">
   <clientDependency xmi:idref="931b8ae1-d253-419e-b10f-57f738062382" />
```

```
    <ownedAttribute xmi:id="d00116cc-f948-482b-a8ff-7f906be1b654" name="Property1"
        xmi:type="uml:Property" type="581302c7-e61b-46f1-bebe-555196beeaf7" visibility="
        public" isDerived="false" isID="false" isReadonly="false" isUnique="true" />
  </packagedElement>
  <packagedElement xmi:id="931b8ae1-d253-419e-b10f-57f738062382" xmi:type="
      uml:Dependency">
    <supplier xmi:idref="8ded957d-048d-4a0c-8183-f06c87ef3181" />
    <client xmi:idref="2d2ef8bb-6029-4222-b546-cf132aa6808e" />
  </packagedElement>
  <packagedElement xmi:id="c74efc66-6972-41da-98e2-e5077ff0d0fd" name="
      ExecutionEnvironment1" xmi:type="uml:ExecutionEnvironment" visibility="public" isAbstract
      ="false">
    <clientDependency xmi:idref="9e64e073-4020-419a-afe1-eaf0b76ab2a4" />
  </packagedElement>
  <packagedElement xmi:id="9e64e073-4020-419a-afe1-eaf0b76ab2a4" xmi:type="
      uml:Dependency">
    <supplier xmi:idref="8da953f0-ff41-4284-9193-adb21e4583c2" />
    <client xmi:idref="c74efc66-6972-41da-98e2-e5077ff0d0fd" />
  </packagedElement>
  <packagedElement xmi:id="8ded957d-048d-4a0c-8183-f06c87ef3181" name="Node1"
      xmi:type="uml:Node" visibility="public" isAbstract="false" />
  <packagedElement xmi:id="26e70823-4db6-4396-aeb1-496e7579527c" name="Datatypes"
      xmi:type="uml:Package" visibility="public">
    <packagedElement xmi:id="581302c7-e61b-46f1-bebe-555196beeaf7" name="Integer"
        xmi:type="uml:DataType" visibility="public" />
    <packagedElement xmi:id="824837ae-78ba-4411-a94d-82d4a234a111" name="Boolean"
        xmi:type="uml:DataType" visibility="public" />
    <packagedElement xmi:id="c58353a9-d70b-4222-b12c-530e0f83215f" name="String"
        xmi:type="uml:DataType" visibility="public" />
    <packagedElement xmi:id="ae4e78ac-e74a-41c9-ac7b-3b4232ae540f" name="
        UnlimitedNatural" xmi:type="uml:DataType" visibility="public" />
    <packagedElement xmi:id="b20f1d23-d5bb-4357-8033-3dea0721e132" name="Object"
        xmi:type="uml:DataType" visibility="public" />
  </packagedElement>
 </uml:Model>
</xmi:XMI>
```

Výpis 12: Výstupní soubor XMI generovaný z výpisu č.4

```
<?xml version="1.0" encoding="utf-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://
    schema.omg.org/spec/UML/2.1.1">
 <xmi:Documentation contact="tomas.huplik@gmail.com" exporter="XMI Factory" exporterVersion
     ="1" />
 <uml:Model xmi:id="6964b68f-3a1b-4af6-8e1b-3b183734dfac" name="TestModel" xmi:type="
     uml:Model" visibility="public">
  <packagedElement xmi:id="f8c85c3b-db3c-4a78-bb1a-cbd976e54ec7" name="Use case 1"
      xmi:type="uml:UseCase" visibility="public">
   <extend xmi:id="bca9b271-9f58-4960-8298-78502459487e" xmi:type="uml:Extend"
       extendedCase="8c6943af-d876-4970-8092-0e9f0f3eddff">
    <extension xmi:idref="f8c85c3b-db3c-4a78-bb1a-cbd976e54ec7" />
   </extend>
  </packagedElement>
```

```
<packagedElement xmi:id="8c6943af-d876-4970-8092-0e9f0f3eddff" name="Use case 2"
    xmi:type="uml:UseCase" visibility="public" />
<packagedElement xmi:id="55416e93-5ec2-405b-a9e7-ea3fbb46c347" name="Use case 3"
     xmi:type="uml:UseCase" visibility="public">
  <include xmi:id="226507ee-1368-4f62-a982-cd7f2a5e828b" xmi:type="uml:Include">
    <addition xmi:idref="747ce9e2-c57f-4051-a46d-20cf827fa2e3" />
  </include>
</packagedElement>
<packagedElement xmi:id="747ce9e2-c57f-4051-a46d-20cf827fa2e3" name="Use case 4"
    xmi:type="uml:UseCase" visibility="public">
  <generalization xmi:id="da8376ec-c2c4-4925-8590-df0879d82414" xmi:type="
      uml:Generalization" isSubstitutable="true" general="f96e0551-7a96-461e-9e4a-
      e89f97e92b7a" />
</packagedElement>
<packagedElement xmi:id="f96e0551-7a96-461e-9e4a-e89f97e92b7a" name="Use case 5"
     xmi:type="uml:UseCase" visibility="public" />
<packagedElement xmi:id="9a969d15-4090-4318-882a-013e4708fe5d" name="Actor1"
    xmi:type="uml:Actor" visibility="public" />
<packagedElement xmi:id="23a4f761-3daa-4c08-bf7c-cb3190f31675" name="" xmi:type="
    uml:Association" visibility="public">
  <memberEnd xmi:idref="18c7660e-7d30-4e4f-bd97-cad00bada70d" />
  <memberEnd xmi:idref="1701db49-166a-4210-88b0-1bdddc01c500" />
  <ownedEnd xmi:id="1701db49-166a-4210-88b0-1bdddc01c500" name="Actor1" xmi:type
      ="uml:Property" type="9a969d15-4090-4318-882a-013e4708fe5d" visibility="public"
      isDerived="false" isID="false" isReadonly="false" isUnique="true">
    <association xmi:idref="23a4f761-3daa-4c08-bf7c-cb3190f31675" />
  </ownedEnd>
  <ownedEnd xmi:id="18c7660e-7d30-4e4f-bd97-cad00bada70d" name="Use case 2"
      xmi:type="uml:Property" type="8c6943af-d876-4970-8092-0e9f0f3eddff" visibility="
      public" isDerived="false" isID="false" isReadonly="false" isUnique="true">
    <association xmi:idref="23a4f761-3daa-4c08-bf7c-cb3190f31675" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:id="877b8a44-2a68-4764-a705-2d1d4f231a42" name="" xmi:type="
    uml:Association" visibility="public">
  <memberEnd xmi:idref="552fd21e-4fe8-4e8a-ba03-2eb285423b8c" />
  <memberEnd xmi:idref="89e744dd-2bd0-423f-a515-5f68b8de0d59" />
  <ownedEnd xmi:id="89e744dd-2bd0-423f-a515-5f68b8de0d59" name="Actor1" xmi:type=
      "uml:Property" type="9a969d15-4090-4318-882a-013e4708fe5d" visibility="public"
      isDerived="false" isID="false" isReadonly="false" isUnique="true">
    <association xmi:idref="877b8a44-2a68-4764-a705-2d1d4f231a42" />
  </ownedEnd>
  <ownedEnd xmi:id="552fd21e-4fe8-4e8a-ba03-2eb285423b8c" name="Use case 5"
      xmi:type="uml:Property" type="f96e0551-7a96-461e-9e4a-e89f97e92b7a" visibility="
      public" isDerived="false" isID="false" isReadonly="false" isUnique="true">
    <association xmi:idref="877b8a44-2a68-4764-a705-2d1d4f231a42" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:id="2cc442fb-b97f-4c55-9091-c65649ccbe43" name="Actor2"
    xmi:type="uml:Actor" visibility="public">
  <generalization xmi:id="eae1ed8d-66bc-4762-b7ba-bfc28a9adc77" xmi:type="
      uml:Generalization" isSubstitutable="true" general="9a969d15-4090-4318-882a-013
      e4708fe5d" />
</packagedElement>
```

```
    <packagedElement xmi:id="730d657e−110a−4505−862d−5fde9e323cb6" name="Datatypes"
        xmi:type="uml:Package" visibility="public">
      <packagedElement xmi:id="581302c7−e61b−46f1−bebe−555196beeaf7" name="Integer"
          xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="824837ae−78ba−4411−a94d−82d4a234a111" name="Boolean"
          xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="c58353a9−d70b−4222−b12c−530e0f83215f" name="String"
          xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="ae4e78ac−e74a−41c9−ac7b−3b4232ae540f" name="
          UnlimitedNatural" xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="b20f1d23−d5bb−4357−8033−3dea0721e132" name="Object"
          xmi:type="uml:DataType" visibility="public" />
    </packagedElement>
  </uml:Model>
</xmi:XMI>
```

Výpis 13: Výstupní soubor XMI generovaný z výpisu č.5

```
<?xml version="1.0" encoding="utf−8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://
    schema.omg.org/spec/UML/2.1.1">
  <xmi:Documentation contact="tomas.huplik@gmail.com" exporter="XMI Factory" exporterVersion
      ="1" />
  <uml:Model xmi:id="a7c91490−9948−4f80−ba51−a8531fb0a0dd" name="TestModel" xmi:type="
      uml:Model" visibility="public">
    <packagedElement xmi:id="5f6107a7−6534−4e79−bece−4978e22b03e9" name="Class1"
        xmi:type="uml:Class" visibility="public" isAbstract="false">
      <clientDependency xmi:idref="ccfc6d6e−53c2−45a4−9163−b9da51902df8" />
      <ownedAttribute xmi:id="e57e6e1d−7f77−433a−99ea−8635ee2c76b8" name="Property1"
          xmi:type="uml:Property" type="91c48f46−2453−4b68−aad0−ac12d8e1faaf" visibility="
          public" isDerived="false" isID="false" isReadonly="false" isUnique="true" />
    </packagedElement>
    <packagedElement xmi:id="ccfc6d6e−53c2−45a4−9163−b9da51902df8" xmi:type="
        uml:Dependency">
      <supplier xmi:idref="808f88a9−b20c−498c−b096−ca97b4cf068a" />
      <client  xmi:idref="5f6107a7−6534−4e79−bece−4978e22b03e9" />
    </packagedElement>
    <packagedElement xmi:id="808f88a9−b20c−498c−b096−ca97b4cf068a" name="Class2"
        xmi:type="uml:Class" visibility="public" isAbstract="false">
      <ownedAttribute xmi:id="f46d4b85−0ba6−47e9−b939−b030c3a2e12c" name="Property2"
          xmi:type="uml:Property" type="91c48f46−2453−4b68−aad0−ac12d8e1faaf" visibility="
          public" isDerived="false" isID="false" isReadonly="false" isUnique="true" />
    </packagedElement>
    <packagedElement xmi:id="ef12cff8−867f−4de3−85fa−6927afd66e6c" name="Object1"
        xmi:type="uml:InstanceSpecification" visibility="public" classifier="5f6107a7−6534−4e79−
        bece−4978e22b03e9">
      <slot xmi:id="af16cbcc−9bdc−4b59−ab5b−c60023697879" xmi:type="uml:Slot"
          definingFeature="e57e6e1d−7f77−433a−99ea−8635ee2c76b8">
        <value xmi:id="1f05d718−9cd4−4f8f−913a−600fc43bd514" visibility="public" xmi:type="
            uml:LiteralString" value="concrete value 1" />
      </slot>
    </packagedElement>
```

```
    <packagedElement xmi:id="eb5c4f7a-cdaf-408e-9286-0c9bb6a63214" xmi:type="
        uml:Dependency">
      <supplier xmi:idref="1f37640a-3b31-49a9-aec9-91e5ee5fea5a" />
      <client  xmi:idref="ef12cff8-867f-4de3-85fa-6927afd66e6c" />
    </packagedElement>
    <packagedElement xmi:id="1f37640a-3b31-49a9-aec9-91e5ee5fea5a" name="Object2"
        xmi:type="uml:InstanceSpecification" visibility="public" classifier="808f88a9-b20c-498c-
        b096-ca97b4cf068a">
      <slot xmi:id="a8ff47b0-43f9-42ff-a402-30ed6b6c4058" xmi:type="uml:Slot"
          definingFeature="f46d4b85-0ba6-47e9-b939-b030c3a2e12c">
        <value xmi:id="a1e1ffaf-91bc-4ccd-82d9-eb602cf6a323" visibility="public" xmi:type="
            uml:LiteralString" value="concrete value 2" />
      </slot>
    </packagedElement>
    <packagedElement xmi:id="1f21d5a3-e40f-4451-a572-f15eef502bb9" name="Datatypes"
        xmi:type="uml:Package" visibility="public">
      <packagedElement xmi:id="11ab9896-51af-434a-ac08-97c8ddd82162" name="Integer"
          xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="08bc1673-564b-4e98-ae74-13bcb8bfd264" name="Boolean"
          xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="91c48f46-2453-4b68-aad0-ac12d8e1faaf" name="String"
          xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="702b983f-dfdd-4f1a-b485-6867a311bb8f" name="
          UnlimitedNatural" xmi:type="uml:DataType" visibility="public" />
      <packagedElement xmi:id="9cf3edb3-abb1-42df-a7d6-99ed4f017bc0" name="Object"
          xmi:type="uml:DataType" visibility="public" />
    </packagedElement>
  </uml:Model>
</xmi:XMI>
```

Výpis 14: Výstupní soubor XMI generovaný z výpisu č.6